RITA Intelligent Transportation Systems Joint Program Office

Security Credential Management System Design

Security system design for cooperative vehicle-to-vehicle crash avoidance applications using 5.9 GHz Dedicated Short Range Communications (DSRC) wireless communications

www.its.dot.gov/index.htm
Draft Report — April 13, 2012
publication number

U.S. Department of Transportation
Research and Innovative Technology Administration

Produced by Walton Fehr
ITS Joint Program Office
Research and Innovative Technology Administration
U.S. Department of Transportation

Notice

Technical Report Documentation Page

1. Report No. **FHWA-JPO-**	**2. Government Accession No.**	**3. Recipient's Catalog No.**
4. Title and Subtitle Security Credential Management System Design Security system design for cooperative vehicle-to-vehicle crash avoidance applications using 5.9 GHz Dedicated Short Range Communications (DSRC) wireless communications		**5. Report Date** April 13, 2012 **6. Performing Organization Code**
7. Author(s)		**8. Performing Organization Report No.**
9. Performing Organization Name And Address		**10. Work Unit No. (TRAIS)** **11. Contract or Grant No.**
12. Sponsoring Agency Name and Address		**13. Type of Report and Period Covered** **14. Sponsoring Agency Code**
15. Supplementary Notes		
16. Abstract		

17. Key Words	**18. Distribution Statement**		
19. Security Classif. (of this report)	**20. Security Classif. (of this page)**	**21. No. of Pages**	**22. Price**

Form DOT F 1700.7 (8-72) **Reproduction of completed page authorized**

Preface/ Acknowledgements

This work is based on CAMP VSC3 – Interoperability Issues of Vehicle-to-Vehicle Based Safety System Project (V2V-Interoperability) – Task 5: Security Management - Subtask 2: Security System Design Specification, September 14, 2011.

It is intended for use during the Safety Pilot Model Deployment.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Definitions

CAMP	Crash Avoidance Metrics Partnership
ITS	Intelligent Transportation Systems
LA	Linkage Authority
LCM	Local Certificate Management
OTA	Over-the-air
RA	Registration Authority
USDOT	United States Department of Transportation
VII	Vehicle Infrastructure Integration
V-V or V2V	Vehicle-to-Vehicle
OBE or OBU	On Board Equipment/Unit
RSE or RSU	Road Side Equipment/Unit
P1609	IEEE P1609 Standard
PSID	Provider Service Identifier
J2735	SAE J2735 DSRC Message Set
BSM	Basic Safety Message (BSM) "HeartBeat"
PKI	Public Key Infrastructure
CRL	Certificate Revocation List
CRT	Certificate (Cert for short)
CSR	Certificate Signing Request
SFAD	Security Framework Access Device (external interface to PKI)
VoD	Verify on Demand
Seed key	A public/private keypair which is used as input to generate another public/private keypair. For example: when generating a request for an implicit cert, the requester generates a seed key which the CA uses to derive the final public key. Also for example: in the butterfly keys model, the requester generates seed keys which are expanded into encryption keys and into intermediate signing keys using the

	expansion function.
Expansion function	The function used in the butterfly keys model to derive a series of encryption keys from the seed encryption key, or to derive a series of intermediate signing keys from the seed signing key.
Batch	A collection of all the short-lived certs for a particular time period, encrypted as a batch by the RA.

1. Introduction

This document describes a security system design for cooperative vehicle-to-vehicle crash avoidance applications using 5.9 GHz Dedicated Short Range Communications (DSRC) wireless communications [1].

1.1 Relevant DSRC Standards

For the DSRC standards (e.g., IEEE 802.11p, 1609.2-4, SAE J2735), please refer to IEEE 802.11p [2]; IEEE P1609 [4], [5], [6]; and SAE DSRC J2735 [3] Basic Safety Message (BSM). The concept of operations is defined to verify good messages, protect against attacks, identify misbehaving OBEs, mitigate attacks and ensure privacy as defined in SAE J2945-1.

1.2 Outline

Sections 1-12 describe a security credential management system for cooperative vehicle-to-vehicle crash avoidance applications.
Appendix A: Certificate File Format describes a certificate file format to store certificates and corresponding private keys.
Appendix B: Security Profiles describes the security profile for all secure communication types applied in Safety Pilot Model Deployment.
Appendix C: Connection requirements between LCM and Server clarifies requirements regarding the LCM establishing and closing connections with the server, according to the various phases of the message protocols.

2. Architecture Overview

This section will give a general model of the system setup.

Two Linkage Authorities (LA1 and LA2) will provide linkage values to the Registration Authority (RA). The RA provides certificates with the support of the Certificate Authority (CA).

There is an OBE that request certificates from the RA. The OBE broadcast V2V safety messages to other OBE, however, this mechanism is out of scope here. The OBE uses a local certificate management (LCM) module to communicate to the RA. The LCM and RA communicate using the previously defined over-the-air message formats. For the purposes of this document, it is assumed that there is a reliable, but not necessarily permanent, connection available between LCM and RA. Such a connection could be provided by cellular connection, Wi-Fi connection, or by road-side equipment (RSE) via DSRC. For ease of exlanation, the details are transparent and it is assumed that a security framework access device (SFAD) is providing the connection between LCM and RA. An overview is shown in Figure 1.

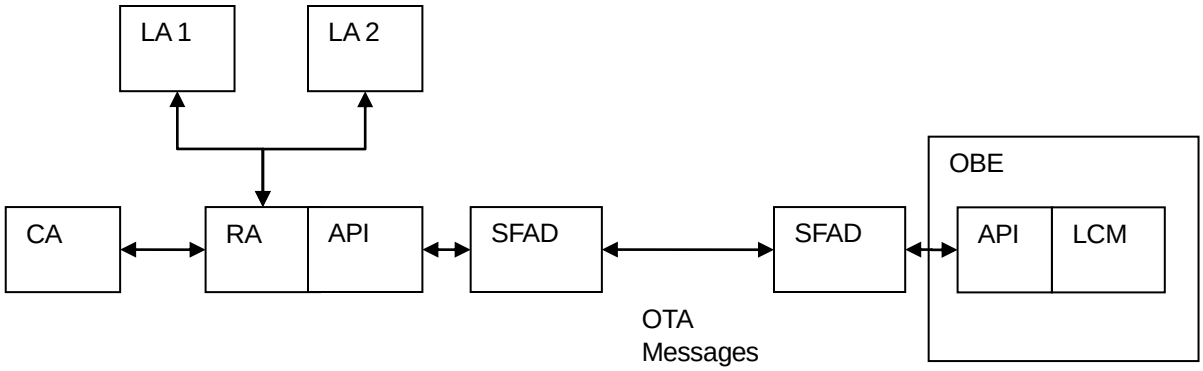


Figure 1: Architecture Overview

3. Certificate Format

3.1 Linked Identifiers

All message certificates (short-term and fall-back message certificates) are imprinted with a linked identifier that allows efficient revocation. The linked identifier is encrypted, and the encrypted value shall be different per certificate. Also retrospective unlinkability shall be preserved: if a node is revoked, it shall not be possible to link certificates that were valid in the past.

It is assumed that all short-term certificates are valid for a globally fixed time period *l*. The server might adjust this value for all short-term certificates that are issued in the future. Fall-back certificates shall also use a globally fixed time validity.

3.1.1 Generation of a message certificate identifier

In the general case, a vehicle can have up to *n* identifiers valid at a given time.

The CA generates the set of certificate IDs {*CertID(w, i, j)*} for vehicle *w* at time period (*i, j)* (where j denotes a sub-period and i consists of n sub-periods, e.g. i describes days and j describes intervals) as follows:

1. Select random *s(w, 0)* and calculate *s(w, 1) = hash(s(w, 0))*
2. Compute *CertID(w, 1, j) =* $Enc_{s(w,\,1)}(j)$ for *j = 1, …, n*
3. For each time period (i, j)
    a. Calculate s(w, i+1) = hash(s(w, i))
    b. Compute *CertID(w, i, j) =* $Enc_{s(w,\,i)}(j)$ for *j = 1, …, n*

For the current system, there will be one certificate per time period (i, j). Otherwise another counter *r* needs to be introduced. If certificates validity starts every 5 minutes with a 30 second overlap, then i describes days, j describes 5:30 minute intervals (with 30 seconds overlap), and n = 288.

Time periods (i, j) are defined that i describes days and j time periods within days. A day always starts at midnight. Time periods (i, j) are defined globally, starting January 1st, 2011, UTC. E.g., (0, 0) describes the time period January 1st, 2011, 00:00:00 – 00:05:29, (0,1) describes January 1tst, 2011, 00:05:00 – 00:10:29, and (3, 3) describes the time of January 4th, 2011, 00:15:00 - 00:20:29.

SHA-256 shall be used for hash *H()*, and AES-128 (ECB) shall be used for *Enc()*.$CertID_{i,r}$ shall be the 10 least significant bytes of $Enc(r, s_i)$, and values s are 128-bit keys. The values s are used as key of AES, thus making it impossible for any instance to compromise the results

CertID by breaking the AES encryption. If we assume around 2^{28} vehicles, 2^{24} (around 5%) revoked OBEs per year, and revoked OBEs are placed in the CRL for one year. Then for a 10 byte CertID, the likelihood of a false positive on the CRL (a benign OBE using the same CertID as a revoked OBE for a given time period) in the system are one time period in $2^{80-28-24} = 2^{28}$ which is about one false positive in the system every 2,550 years. Smaller revocation rates and higher rate of false positive rates might allow to reduce the size of the CertID to 8 bytes.

The CA stores values s(w, 0), and the time (t_{start}, t_{end}) when the first certificate of the set is valid.

3.1.2 Generation of fall-back certificate identifier

It seems the easiest way of doing this is to consider fall-back certificates separately. Then to revoke an OBE, we need two entries in the CRL: one for regular certs, another for the fall-back certificate(s). Fall-back certificates use values (i, j) with $i \geq 2^{31}$ and j=0.

3.1.3 Revocation: Server

First, the server determines the current time interval (i, j) at which the device was revoked. The server then adds (s(w, i), i, max_i) to the CRL. The value max_i describes the time period until the CRL entry is valid. This mechanism applies to both short-term and fall-back certificates.

3.1.4 Revocation: LCM

Before a time interval starts, the LCM calculates the *certID* of revoked nodes for that time interval (e.g. by updating s_i and calculating $Enc(r, s_i)$). Thus the LCM holds at all times a (dynamic) list of revoked certificates for the current time interval. The list of revoked certificates might comprise entries for short-term and for fall-back certificates.

3.2 Implicit Certificates

Implicit certificates according to IEEE 1609.2 D9.3 are used as message certificates.

3.3 Fall-Back Certificates

Fall-back certificates are always accessible by LCM, i.e. LCM does not need to request a decryption key for fall-back certificates.

2.4 Certificate Batches

A large number of encrypted certificates is loaded at a time. This will be called batch-mode. A batch is the set of certificates that is encrypted by the RA using an individual key. For

instance, if certificates worth one year are loaded, then there might be 12 batches each including certificates worth one month.

3.5 Elliptic Curve Points

All elliptic curve points in these messages shall be expressed in compressed form.

3.6 Certificate File Format

A file format to store certificates and private keys is specified in Appendix A: Certificate File Format.

4. CRL Management

In this design, an extended IEEE 1609.2 CRL format is used to account for Linked Identifiers. The RA will maintain an "internal" CRL (listing revoked CSR certs) to validate LCM requests, and the CA will maintain the public revoked cert information repository. This shall include maintaining, for each revoked unit, the CertID sequences for that unit and the time at which the unit can be removed from the list of revoked units. It shall also include the full set of individual CRLs sent to date.

Delta CRLs (ie CRLs that list only units that are newly revoked since the previous CRL) will be used and sequentially numbered. The CA will issue a new Delta CRL at configurable intervals. The CA will maintain both Delta CRLs and a whole CRL that lists all of the currently revoked units with unexpired certs.

5. OTA Message Formats

5.1 Extensions to Existing 1609.2 Messages

This section defines extensions to the 1609.2 secured message types necessary to support the security management functionality defined by CAMP. All messages sent shall be encapsulated within 1609Dot2Messages with the modifications noted below.

5.1.1 ContentType

The ContentType enumerated type shall be extended as follows:

contentType	Value
crl_req	236
crl_req_error	237
misbehavior_report_req	238
misbehavior_report_ack	239
cert_bootstrap_req	240
cert_bootstrap_cfm	241
cert_bootstrap_ack	242
anonymous_cert_request_req	243
anonymous_cert_request_cfm	244
anonymous_cert_request_status_req	245
anonymous_cert_request_status_cfm	246
sig_enc_cert	247
certificate_and_private_key_reconstruction_value	248
anonymous_cert_response_ack	249
anonymous_cert_decryption_key_req	250
anonymous_cert_decryption_key_cfm	251
anonymous_cert_decryption_key_error	252
anonymous_cert_decryption_key_ack	253

symmetric_encrypted	254

5.1.2 1609Dot2Message

The 1609Dot2Message structure shall be extended as follows:

```
    struct  {
                uint8        protocol_version;
                ContentType  type;
                select (type) {
                case unsecured :
                    opaque           message<var>;
                case signed, signed_partial_payload,
                     signed_external_payload:
                    SignedMessage    signed_message;
                case signed_wsa:
                    SignedWsa        signed_wsa;
                case encrypted :
                    EncryptedMessage encrypted_message;
                case crl_request :
                    CrlRequest       crl_request;
                case crl :
                    Crl              crl;
// begin new material
                case cert_bootstrap_req:
                    CertificateRequest cr;
                case cert_bootstrap_cfm :
                    BootStrapConfirm bsc;
                case cert_bootstrap_ack:
                    BootStrapAcknowledgement bsa;
                case anonymous_cert_request_cfm
                    SignedMessage cert_req_cfm;
                case symmetric_encrypted:
                    SymmetricEncryptedMessage sem;
// end new material
                unknown:
                    opaque           message<var>;
                }
            } 1609Dot2Message;
```

5.1.3 ToBeEncrypted

The ToBeEncrypted message structure shall be extended as follows:

```
    struct {
                ContentType                  type;
                select(type) {
                case unsecured:
                    opaqueExtLength          plaintext;
                case signed, signed_external_payload,
                     signed_partial_payload :
                    SignedMessage            signed_message;
```

```
                case certificate_request :
                    CertificateRequest        request;
                case certificate_response :
                    ToBeEncryptedCertificateResponse
                                              response;
                case certificate_request_error:
                    ToBeEncryptedCertificateRequestError
                                              request_error;
                case crl_request :
                    CrlRequest                crl_request;
                case crl :
                    Crl                       crl;
                case certificate_response_acknowledgment:
                    ToBeEncryptedCertificateResponseAcknowledgment
                                              ack;
// begin new material

                case anonymous_cert_request_req:
                case anonymous_cert_request_status_req:
                case anonymous_cert_response_ack:
                case anonymous_cert_decryption_key_req:
                case anonymous_cert_decryption_key_ack:
                case misbehavior_report_req:
                case misbehavior_report_ack:
                case crl_req:
                     SignedMessage                sm;
                case sig_enc_cert:
                     SignedMessage sec<var>;
                case anonymous_cert_request_status_cfm:
                     AnonymousCertRequestStatusCfm acbrsc;
                case anonymous_cert_decryption_key_cfm:
                     AnonymousCertDecryptionKeyCfm ascdkrc;
                case anonymous_cert_decryption_key_error
                     ToBeEncryptedDecryptionKeyRequestError err;
                case certificate_and_private_key_reconstruction_value:
                      CertificateAndPrivKeyReconstructionValue cpkrv;
                case crl_req_error:
                     ToBeEncryptedCrlReqError tbecre;

// end new material
                unknown:
                    opaque                    message<var>;
                }
           }   ToBeEncrypted;
```

5.1.4 SignedMessage

The SignedMessage type shall be extended as follows:

```
    struct {
                extern ContentType   type;
```

```
            SignerIdentifier     signer;
            select (type) {
            case signed, signed_partial_payload,
                signed_external_payload:
                ToBeSignedMessage             unsigned_message;
            case anonymous_cert_request_req:
                ToBeSignedAnonymousCertRequestReq       acrbrr;
            case anonymous_cert_request_cfm;
                AnonymousCertRequestConfirm             acrc;
            case anonymous_cert_request_status_req:
                ToBeSignedAnonymousCertRequestStatusReq acrsr;
            case sig_enc_cert:
                ToBeSignedEncCert                       tbsec;
            case anonymous_cert_response_ack:
                ToBeSignedAnonymousCertResponseAck      tbsaca;
            case anonymous_cert_decryption_key_req:
                ToBeSignedAnonymousCertDecryptionKeyReq acdkr;
            case anonymous_cert_decryption_key_ack:
                ToBeSignedAnonymousCertDecryptionKeyAck tbsacdka;
            case misbehavior_report_req:
                ToBeSignedMisbehaviorReportReq          tbsmrr;
            case misbehavior_report_ack:
                ToBeSignedMisbehaviorReportAck          tbsmra;
            case crl_req:
                ToBeSignedCrlReq                        tbscr;
            }
            Signature            signature;
        } SignedMessage;
```

5.1.5 Certificates: SubjectType, SubjectTypeFlags

```
    enum   {  message_anonymous(0),
              message_identified_not_localized (1),
              message_identified_localized (2),
              message_csr (3)
              wsa (4),
              wsa_csr (5),
              message_ca(6), wsa_ca (7), crl_signer(8),
// new material
              message_ra (9)
// end new material
              ...
              root_ca (255),
              (2^8-1)
           }  SubjectType;

    flags  {  message_anonymous(0),
              message_identified_not_localized (1),
              message_identified_localized (2),
              message_csr (3)
              wsa (4),
```

```
            wsa_csr (5),
            message_ca(6), wsa_ca (7), crl_signer(8),
// new material
            message_ra (9)
// end new material
          }  SubjectTypeFlags;
```

A certificate with SubjectType value message_ra is identical in syntax to a certificate with SubjectType value message_ca. In other words, in every 1609.2 structure that has a select() or if_any_set() statement with an entry for case message_ca, that should be read as case message_ca message_ra. This applies to ToBeSignedCertificate, CertSpecificData, RootCaScope, CertRequestSpecificData.

An RA certificate may not be used to sign another certificate, while a CA certificate may.

5.1.6 RootCAScope

The RootCAScope type shall be extended as follows:

```
    struct {
            uint8               name<var>;
            RootCAScope
    permitted_subject_types;
            if_any_set (permitted_subject_types, message_ca,
                message_csr, message_identified_localized,
                message_identified_not_localized,
                message_anonymous
// new material
            ,message_ra
// end new material
) {
                RootCAScope
         message_permissions;
            }
            if_any_set (permitted_subject_types, wsa_ca,
                wsa_csr, wsa) {
                RootCAScope wsa_permissions;
            }
            if_value_not_in (permitted_subject_types,
                message_ca,
                message_csr, message_identified_localized,
                message_identified_not_localized,
                message_anonymous, wsa_ca,
                wsa_csr, wsa
// new material
            ,message_ra, crl_signer
// end new material

) {
```

```
            opaque            other_permissions<var>;
        }
        RootCAScope  region;
    } RootCAScope;
```

5.1.7 CertificateRequestErrorCode

The CertificateRequestErrorCode shall be extended as follows:

```
enum  {  csr_cert_verification_failure(0), csr_cert_expired(1),
         csr_cert_revoked(2), csr_cert_unauthorized(3),
         request_denied(4), csr_cert_unknown (5),
         canonical_identity_unknown (6),
         certificate_response_not_ready (7),
         cert_set_start_time_in_past (8),
         cert_set_start_time_too_far_in_future (9),
         cert_set_end_time_too_close_to_start_time (10),
         cert_set_end_time_too_far_from_start_time (11),
         requested_smaller_than_minimum_batch_duration (12),
         requested_larger_than_maximum_batch_duration (13),
         requested_past_decryption_keys (14),
         requested_far_future_decryption_keys (15),
         invalid_signature (16),
         invalid_request_hash (17),
         invalid_response_encryption_key(18),
         invalid_status (19),
         invalid_algorithm (20),
         current_time_in_past(21),
         current_time_in_future(22),
         ... (255)
      } CertificateRequestErrorCode;
```

5.2 Bootstrapping

5.2.1 Overview

Before bootstrapping, the CA does the following with mechanisms which may or may not be the same as the mechanisms in this document:

- Generates a cert for the RA and makes it available to the RA.
- Generates a cert for the CA and makes it available to the RA

On bootstrap, the LCM does the following:

- Generates a seed key for the CSR cert (the implicit cert). This is expressed in compressed form.
- Generates a ToBeSignedCertificateRequest for the CSR cert
- Signs it with the seed private key to form a CertificateRequest

- Encapsulates it in a 1609Dot2Message of type cert_bootstrap_req and sends it to the RA.

The RA sends the cert request to the CA.

The CA forms the implicit certificate for the CSR cert and returns it to the RA.

The RA

- Forms a ToBeEncryptedCertificateResponse containing the implicit cert, the CA cert, and the private key reconstruction value
- Forms a BootStrapConfirm containing the ToBeEncryptedCertificateResponse and the RA Certificate
- Encapsulates the BootStrapConfirm in a 1609Dot2Message and sends it to the LCM.

The LCM:

- Sends a BootStrapAck.
- Stores the RA certificate and CA certificate
- Reconstructs the private key for the implicit CSR cert using the reconstruction value and stores that private key and cert.

Note: the bootstrap mechanism must be executed in a secure environment. The Security Server must be able to recognize whether an LCM is authorized to request credentials. The details of how this process is performed will depend upon policy decisions. Therefore in this design it is assumed that the bootstrap is executed in a secure environment, that OBE vendors are trustworthy, and that OBE vendors only execute bootstrap for authorized LCMs. In order to control the bootstrap execution, the Security Server must explicitly enable bootstrap process by setting the proper configuration flag, or by human intervention.

Note: in this design, we implement the initialization and re-initialization of an LCM (after revocation) as a bootstrap. We implement pre-loading of an LCM as bootstrap + request of certificates:

re-init (after revocation) = init = bootstrap

pre-loading = bootstrap + certificate request

5.2.2 LCM → RA (Request)

The LCM requests a bootstrap with a 1609Dot2Message of type cert_bootstrap_req containing a CertificateRequest. It is assumed that the message is sent within a secure environment and as such this message is not explicitly protected by 1609.2 mechanisms.

Within the structure the fields have the following meaning:

- request is a standard CertificateRequest for a CSR cert in which:
  - info.type shall be self;
  - the signature is generated using the private key corresponding to the public key in the verification_key field.
- In the ToBeSignedCertificateRequest:
  - the subject_type field shall be message_csr;
  - the permitted_subject_types field in the MessageCaScope shall contain only the value message_anonymous;
  - the name field in the MessageCaScope shall contain a unique identifier for the LCM;
  - the permissions field in the MessageCaScope shall contain the PSIDs from which the originator of this intends to send application messages;
  - the region field in the MessageCaScope shall be of type NONE;
  - the content flags use_start_validity shall be set and the start_validity field shall be filled in with 00:00:00 am, Jan 1, 2011;
  - the field expiration shall be set to 11:59:59 pm, June 30, 2013
  - the field verification_key shall contain the seed public verification key of the sender. This shall be an ECDSA-256 key in compressed form.
  - the content flag encryption_key shall not be set;
  - the field response_encryption_key may contain a freshly generated ECIES-NISTp256 encryption public key or may be all NULLs – this field is not processed at the receiving end so it doesn't matter what's in it.

5.2.3 RA → LCM (Confirm)

```
struct {
      Certificate        ra_certificate;
      ToBeEncryptedCertificateResponse resp;
} BootStrapConfirm;
```

This type is used to return a bootstrap response to the requesting LCM. It shall be encapsulated within a 1609Dot2Message of type cert_bootstrap_cfm. It is not encrypted.

- the ra_certificate field shall contain the RA's public encryption and verification key. The LCM shall use these keys for messaging the RA. For this design, The RA certificate shall have the following properties:
  - It shall be an explicit certificate (version_and_type = 2)
  - The signing key shall be an ECDSA-256 key given in compressed form.
  - subject_type shall be message_ra.
  - It shall have expiration time set to June 30, 2013 and crl_series set to 1.
  - It shall not have the ContentFlag values use_start_validity or content_is_duration set.

    - It shall have the ContentFlag value encryption_key set and contain an encryption key for ECIES-256 given in compressed form.
    - All permissions fields within the MessageCaScope will be identical to the equivalent fields in the CA certificate, defined below, with the exception that the SubjectFlags shall be 83 ff (anonymous, identified not localized, identified localized, message_csr, wsa, message_ca, wsa_ca, crl_signer, message_ra, message_ra)
- The certificate_chain field in resp shall contain a chain of length 2, consisting of the CA cert (which will have subject_type equal to root_ca) and the requester's CSR cert.
    - The CSR cert is not guaranteed to have the same values in any fields as were provided in the corresponding field in the request.
    - The CA cert shall have the following properties:
        - It shall be an explicit certificate (version_and_type = 2)
        - subject_type shall be root_ca.
        - It shall have expiration time set to June 30, 2013 and crl_series set to 1.
        - It shall not have the ContentFlag values use_start_validity or content_is_duration set.
        - The signing key shall be an ECDSA-256 key given in compressed form.
        - It shall have the ContentFlag value encryption_key set and contain an encryption key for ECIES-256 given in compressed form.
        - The contents of the RootCaScope shall be:
            - Name: No name provided
            - SubjectFlags encoding: 83 ff (anonymous, identified not localized, identified localized, message_csr, wsa, message_ca, wsa_ca, crl_signer, message_ra, message_ra)
            - Message Permissions: psid: 0x10 – 0x14 (16 -20)
            - Wsa Permissions: psid: 1 priority: 1
            - Geographic Region: None
- the crl_path shall contain a single CRL. It shall be the most current whole CRL. The CRL shall have the format as defined in Section 5.6.3 LCM → RA (Request)
- .
- The flags field in resp shall be the value 0.

5.2.4 LCM → RA (Acknowledgement)

```
struct {
      opaque response_hash[10];
} BootStrapAcknowledgement;
```

This type is used to signify to the RA that the bootstrapping process was successfully completed. It shall be encapsulated within a 1609Dot2Message of type cert_bootstrap_ack.

In this design, the LCM shall generate and send a bootstrap acknowledgement once it receives a BootStrapConfirm. It does not need to perform any checks on the contents of the BootStrapConfirm before sending the ack.

The response_hash field is the low-order ten bytes of the hash of the encoded BootStrapConfirm message contained in the response from the RA to the LCM.

If the RA does not receive a BootStrapAcknowledgement within a configurable timeout period, it shall resend the response.

If the RA receives a BootStrapAcknowledgement with an incorrect response_hash value, it shall resend the response.

If the RA receives an incorrect response_hash three times, it shall stop sending the response and log an error.

5.3 Request Certificates

5.3.1 Overview

- The LCM:
    - Generates a ToBeSignedAnonymousCertRequestReq
    - Signs it with the CSR certificate to create a SignedMessage.
    - Creates a ToBeEncrypted of type anonymous_cert_request_req
    - Encrypts it with the RA's encryption key to create an EncryptedMessage
    - Encapsulates that in a 1609Dot2Message of type encrypted
    - Sends it to the RA
- The RA:
    - Decrypts the request
    - Hashes the request
    - Creates a signed message of (external) type anonymous_cert_request_cfm containing the hash of the request and the signature.
    - Encapsulates that in a 1609Dot2Message of type anonymous_-cert_request_cfm.
    - Sends it to the LCM.

If the signature verifies and the hash is correct, the LCM takes no action. If the signature verifies and the hash is wrong the LCM resends the request. If the signature does not verify then the LCM takes no action.

Meanwhile, the following processing takes place internally at the PKI:

- The RA:
    - Verifies the request and determines whether to issue the cert
    - Uses the seed keys and expansion functions in the request to generate individual cert requests for each cert containing request-specific signing keys and encryption keys.
    - Sends the individual cert requests to the CA, along with (i, j) for each cert.
- The CA, for each certificate
    - Generates the certificate and c
    - Encodes them in a CertificateAndPrivKeyReconstructionValue

- Creates a ToBeEncrypted of type certificate_and_private_key_reconstruction_value
- Encrypts it with the request-specific ECIES key to form an EncryptedMessage
- Creates a ToBeSignedEncCert containing the encrypted certificate and information about i, j, start time, and end time.
- Signs it to form a SignedMessage of (external) type sig_enc_cert.
- Returns the SignedMessage to the RA along with an indication of the cert request that the SigEncCert is associated with (from which the RA can extract i and j) and the start / end times of the cert (this return message is not specified in this document).

After a while, the LCM re-establishes a connection with the PKI and requests status.

- The LCM:
  - Generates a ToBeSignedAnonymousCertRequestStatusReq
  - Signs it to generate a SignedMessage of (external) type anonymous_cert_request_status_req.
  - Creates a ToBeEncrypted of (explicit) type anonymous_cert_request_status_req.
  - Encrypts it with the RA's public key to form an EncryptedMessage
  - Encapsulates it in a 1609Dot2Message of type encrypted
  - Sends it to the RA.
- The RA:
  - If the cert issuance is approved and the certs are ready:
    - For each batch period:
      - Collects the SignedMessages of type sig_enc_cert for that batch.
      - Creates a ToBeEncrypted of type sig_enc_cert and contents SigEncCert<var> containing all the certs for the period
      - Encrypts the ToBeEncrypted with AES-CCM to obtain an AesCcmCiphertext. Stores the key somewhere.
      - Creates an EncryptedCertificateBatch containing the AesCcmCiphertext and the other fields.
    - Aggregates all the EncryptedCertificateBatches to be included in the response, and the fallback certs, into an AnonymousCertRequestStatusCfm, along with the permissions

        - Creates a ToBeEncrypted of type anonymous_cert_request-_status_cfm containing the AnonymousCertRequestStatusCfm.
        - Encrypts with the response_encryption_key from the status request and forms a SymmetricEncryptedMessage
        - Encapsulates the SymmetricEncryptedMessage in a 1609Dot2Message of type symmetric_encrypted.
        - Sends to the LCM.
    - If the cert issuance is not approved or the certs are not ready:
        - Creates and signs a ToBeEncryptedCertificateRequestError as described in 1609.2 with the appropriate error code (see extensions to CertificateRequestErrorCode above)
        - Encrypts it with the response_encryption_key from the status request to form an EncryptedMessage
        - Encapsulates the SymmetricEncryptedMessage in a 1609Dot2Message of type symmetric_encrypted.
        - Sends to the LCM.
- The LCM:
    - Decrypts the message
    - Sends an ACK.
    - Immediately sends a request to decrypt the first batch of certificates (see next section).
    - If appropriate, attempts to decrypt the first fallback cert.

5.3.2 LCM → RA (Request)

```
struct {
      PublicKey           A;
      PublicKey           H;
      uint8               s[16];
      uint8               e[16];
      PsidSspArray        permissions;
      GeographicRegion    region;
      Time32              current_time;
      Time32              start_time;
      Time32              end_time;
      CertificateDuration batch_duration;
      uint32              storage_space_kb;
      CertId8             known_cas<var>
```

```
} ToBeSignedAnonymousCertRequestReq;
```

This type is used to allow the sender to request the anonymous certificates. It shall be signed with the CSR certificate to create a SignedMessage of (external) type anonymous_-cert_request_req. This shall be included within a ToBeEncrypted of (explicit) type anonymous_cert_request_req. This shall be encrypted with the public key of the RA, obtained from its certificate, to produce an EncryptedMessage. This in turn shall be encapsulated within a 1609Dot2Message of type encrypted.

In the SignedMessage:

- the signer.type field shall be certificate and signer.certificate shall be the CSR cert that the requester obtained on bootstrap.
- the signature field shall be the signature, calculated over the encoding of acrbrr, using the private key corresponding to the public verification key in signer.certificate.

In the ToBeSignedAnonymousCertRequestReq:

- the A field shall contain the seed public key for signing. This shall be an ECDSA-256 key given in compressed form.
- the H field shall contain the seed public key for encryption. This shall be an ECIES-256 key given in compressed form.
- the s field shall contain a randomly-generated 16-byte AES key, which will be used to generate the cocoon keys for signing.
- the e field shall contain a randomly-generated 16-byte AES key, which will be used to generate the cocoon keys for encryption.
- The permissions field shall contain the permissions being requested. In this design, it shall contain a single PSID and a NULL SSP for that PSID.
- The region shall be none.
- the current_time field shall contain the current time.
- the start_time field shall contain the time at which the validity period is requested to begin for the short-lived certificates.
- the end_time field shall contain the time at which the validity period is requested to end for the short-lived certificates.
- the batch_duration field shall contain the requested duration of each encrypted batch of certificates.
- the storage_space_kb field shall contain the maximum amount of space (in kb, i.e. blocks of 1024 bytes) that the requester has available to store the requested certs.
- the known_cas field shall contain the CertID8s for all the CAs that the requester recognizes.

The PKI shall prepare certs as follows.

- The PKI shall return at least enough fallback certs to enable the vehicle to operate for up to GCM_T_reload_fallBack (as defined in Section 10. Security Server Configuration Parameters) from current_time.
- The PKI shall return short-lived certs such that the total space taken up by the certs is no more than storage_space_kb and the final cert's start time is no later than the value in the end_time field. The PKI shall always return that number of fallback certs and may at its discretion return less short-lived certs than the maximum.

5.3.3 RA → LCM (Confirm)

```
struct {
      uint8               accept;
      Time32              request_time;
      opaque              request_hash[10];
      select(accept) {
      case 0:
        CertificateRequestErrorCode   reason;
      }
} AnonymousCertRequestConfirm;
```

To confirm receipt of an anonymous certificate request, the RA shall send the OBE a confirmation message. The RA starts by creating an AnonymousCertRequestConfirm. In this structure:

- accept is a 1 if the anonymous_cert_request_req was acceptable to the server otherwise it is a 0
- request_time is the time the request was generated
- request_hash is the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedAnonymousCertRequestReq contained in the request from the OBE to the RA.
- reason is the reason code why the request is not acceptable.

The RA signs this message with its certificate to create a SignedMessage of (external) type anonymous_cert_request_cfm, and encapsulates it within a 1609Dot2Message of (explicit) type anonymous_cert_request_cfm.

If the LCM does not receive a correct confirm (in other words, if it receives no confirm, or only confirms whose signatures do not verify) within a configurable timeout period, it shall regenerate and resend the request, including regenerating A, H, s and e.[1]

[1] Regenerating the request means that the current time field in the AnonymousCertRequest will change, as will the encapsulation within a 1609.2 EncryptedMessage. Therefore, an eavesdropper will not be able to tell the difference between a resend and an initial send.

If the LCM receives a confirm whose signature verifies but which contains an incorrect request_hash value, it shall regenerate and resend the request.

If the LCM receives distinct confirms with an incorrect request_hash value three times in a row (in other words, the signature is valid, the timestamp in each of the three confirm messages is distinct, but the request_hash is wrong), it shall stop trying to send the request and put itself in an appropriate error state (eg logging the error).[2]

If the reason is:

- csr_cert_verification_failure(0), the LCM shall regenerate and resend the certificate request
- csr_cert_expired(1), csr_cert_revoked(2), csr_cert_unauthorized(3), request_denied(4), csr_cert_unknown (5), or canonical_identity_unknown (6), the LCM shall log the error and go into an error state
- cert_set_start_time_in_past (8), cert_set_start_time_too_far_in_future (9), cert_set_end_time_too_close_to_start_time (10), cert_set_end_time_too_far_from_start_time (11), requested_smaller_than_minimum_batch_duration (12), or requested_larger_than_maximum_batch_duration (13), the LCM shall log the message and go into an error state.
- invalid_signature (16), the LCM shall regenerate and resend the certificate request.
- All other error codes are not valid in this case. If the LCM receives an invalid error code, it shall log an error and regenerate and resend the certificate request.

5.3.4 LCM → RA (Status Request)

```
struct {
      Time32                        current_time;
      CertificateProcessingStatus   status;
      opaque                        request_hash[10];
      SymmAlgorithm                 alg;
      opaque                        response_encryption_key [16];
} ToBeSignedAnonymousCertRequestStatusReq;

enum { success (0), first_request(1), got_certs_not_ready(2),
     timeout(3),  err_decrypt (4), .. (255)
} CertificateProcessingStatus;
```

[2] This is the reason for including the current_time in the confirm message. Otherwise, if an attacker could detect an incorrect confirm message, they could replay it three times and this would effectively provide a kill switch for the OBE.

After sufficient time has passed, the LCM can use this type to request that the RA provides the previously requested certificates.

- current_time is the time the status request was generated.
- status is the LCM's understanding of the status. It shall be:
  - first_request if this is the first status request with a given request_hash value;
  - got_certs_not_ready if the last status request with this request_hash value received an error response with CertificateRequestErrorCode value certificate_response_not_ready;
  - timeout if the LCM did not receive a response within the configurable timeout time to the last status request with this request_hash value;
  - err_decrypt if the LCM got an error decrypting the response to the last status request with this request_hash value.
- request_hash is the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedAnonymousCertRequestReq contained in the request from the LCM to the RA
- alg shall be aes_128_ccm.
- the response_encryption_key field shall contain the AES-CCM key to be used to encrypt the response. The LCM shall generate a fresh response encryption key every time it generates a ToBeSignedAnonymousCertRequestStatusReq and shall not (or shall only with negligible probability) repeat response encryption keys.

This is signed and encapsulated in a SignedMessage, which is then put inside a ToBeEncrypted with type anonymous_cert_request_status_req, which is then encrypted with the public key of the RA and encapsulated within a 1609Dot2Message of type encrypted.

The LCM is not assumed to preserve records of any status request other than the most recent one, i.e. the one with the latest current_time value.

5.3.5 RA → LCM (Status Confirm, success)

The RA shall return the anonymous certificates within a 1609Dot2Message of type symmetric_encrypted. This shall contain a SymmetricEncryptedMessage, as follows:

```
struct {
      SymmAlgorithm              alg;
      opaque                     key_id[8];
      AesCcmCiphertext           ciphertext;
} SymmetricEncryptedMessage;
```

In this structure:

- alg is the algorithm used to encrypt the message

- key_id is the low-order 8 bytes of the SHA-256 hash of the key used to encrypt the message. This shall be the response_encryption_key from the corresponding ToBeSignedAnonymousCertRequestStatusReq.
- ciphertext is the encrypted data.

When the ciphertext is decrypted with the key indicated by key_id, the result shall be a ToBeEncrypted with type anonymous_cert_request_status_cfm and contents equal to a message of type AnonymousCertRequestStatusCfm:

```
struct {
      opaque                      request_hash[10];
      PsidSspArray                permissions;
      GeographicRegion            region;
      EncryptedCertificateBatch   short_lived_certs<var>;
      UnencryptedCertificateBlock fallback_certs;
} AnonymousCertRequestStatusCfm;
```

The type AnonymousCertRequestStatusCfm is used by the RA to return the anonymous certificates. In the AnonymousCertRequestStatusCfm:

- request_hash is the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedAnonymousCertRequestReq message contained in the request from the LCM to the RA
- permissions contains the permissions field that appears in every certificate
- region contains the region field that appears in every certificate
- The short_lived_certs field shall contain one or more instances of the data structure EncryptedCertificateBatch, each encapsulating an encrypted batch of short-lived certs.
- The fallback_certs field shall contain a single instance of the data structure UnencryptedCertificateBlock encapsulating a block of fallback certs.

```
struct {
      uint8                       cert_batch_id[8];
      Time32                      start_time;
      Time32                      end_time;
      uint32                      validity_period;
      uint32                      overlap_period;
      AesCcmCiphertext            enc_batch;[3]
} EncryptedCertificateBatch;
```

[3] When these message formats are migrated back to 1609.2 we'll need make this switch on a SymmAlgorithm value, but this definition is fine for now as only AES is defined.

The type EncryptedCertificateBatch contains a batch of encrypted certificates, for example the encryption of all of the certificates for a given month. The fields in the EncryptedCertificateBatch have the following meanings.

- the cert_batch_id field shall identify the batch of certificates for which the request is being processed and shall be used to associate a batch decryption key with the batch to be decrypted. It shall be determined by the RA and shall be unique for every encrypted certificate batch sent from the RA to the LCM.
- the start_time field shall contain the time at which the validity period begins for the first certificate in the batch.
- the end_time field shall contain the time at which the validity period ends for the last certificate in the batch.
- the validity_period field shall contain the validity period in seconds for each certificate in the batch.
- the overlap_period field shall contain the overlap period in seconds for each certificate in the batch.
- the enc_batch field shall contain the ciphertext of the array of encrypted certificates.

When decrypted, the AesCcmCiphertext enc_batch shall contain a ToBeEncrypted of type sig_enc_cert and the contents shall be a collection of SignedMessages (note the "<var>" in the definition in Section 5.1.3 ToBeEncrypted). Each SignedMessage shall contain a single ToBeSignedEncCert, as follows:

```
struct {
      uint32              i;
      uint32              j;
      Time32              expiration;
      CertificateDuration lifetime;
      EncryptedMessage    enc_cert;
} ToBeSignedEncCert;
```

The field enc_cert shall have been encrypted with the ECIES encryption key derived from the seed encryption key H and the expansion function f(i, j).

For short-term certificates, time periods (i, j) are applied as defined in Section 3.1.1 Generation of a message certificate identifier with time period (0, 0) describing the time period starting January 1st, 2011, 00:00:00, UTC.
For fall-back certificates, time periods (i, j=0) are applied with j always set to 0 and $i \geq 231$, and (231, 0) describing the time period starting January 1st, 2011, 00:00:00, UTC.

When enc_cert is decrypted it shall contain a ToBeEncrypted with type certificate_and_private_key_reconstruction_value and contents CertificateAndPrivKeyReconstructionValue, as below:

```
struct {
      Certificate         cert;
      uint8               s[32];
} CertificateAndPrivKeyReconstructionValue;
```

The fields here have the following properties:

- the s field shall contain the private key reconstruction value, calculated as described in Section 8.5 Request Certificates.
- the cert field shall contain the Certificate for a given time period and contains the following fields:
    - the certificate_version_and_type field shall be equal to the integer 3.
    - the unsigned_certificate.subject_type field shall contain the type message_anonymous.
    - the unsigned_certificate.cf field shall have the flags use_start_validity and lifetime_is_duration set and the flag encryption_key not set.
    - the unsigned_certificate.scope field shall be an AnonymousScope. In this field:
        - additional_data shall contain an AnonymousRevocationInformation structure.
        - permissions shall contain a single PSID with a null SSP encoding the permissions for the cert
        - region shall be of type NONE.
    - unsigned_certificate.expiration field shall contain the time at which validity period ends for the certificate.
    - the unsigned_certificate.lifetime field shall contain the duration of a single certificate (encoded value for 5 minutes 30 seconds).
    - the unsigned_certificate.crl_series field shall be the CRL series. For the current design this field shall always have the value 1.
    - the unsigned_certificate.signer_id field shall contain the low-order eight bytes of the hash of the CA certificate.
    - the reconstruction_value field shall be the reconstruction value $C = B_n + c*G$ calculated by the CA as described in Section 8.5 Request Certificates. This shall be an ECPublicKey with algorithm ECDSA-256, given in compressed form.

```
struct {
      int32               i;
      int32               j;
      opaque              cert_id[8];
} AnonymousRevocationInformation;
```

The fields here have the following properties:

- i and j are used to calculate the cert_id from the seed

- cert_id is the cert ID calculated by the CA by XORing the two linkage values

```
struct {
      SignedMessage               sig_enc_cert<var>;
} UnencryptedCertificateBlock;
```

The type UnencryptedCertificateBlock contains an unencrypted block of certificates, for example a set of fallback certificates. (Note that although the block is unencrypted, the individual certificates within the block are encrypted individually). The fields in the UnencryptedCertificateBlock have the following meanings.

- the sig_enc_cert field shall contain the array of individually encrypted certificates.[4] These are SignedMessages of (external) type sig_enc_cert.

NOTE: The LCM shall keep a list of all response_encryption_keys sent in the order in which they were sent. A response is considered "correct" if it decrypts correctly and the signature on the message inside the response verifies. If the LCM receives a correct response using a given response_encryption_key it shall discard any future messages encrypted with that response_encryption_key, to prevent replay attacks.

5.3.6 RA → LCM (Status Confirm, failure)

In the case of a failure, or certificates not ready yet, the RA shall send the LCM a Certificate Request Error using the ToBeEncryptedCertificateRequestError as described in 1609.2 clause 5.3.37.

In the ToBeEncryptedCertificateRequestError, request_hash is the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedAnonymousCertRequestReq message contained in the request from the LCM to the RA.

The ToBeEncryptedCertificateRequestError shall be signed with the RA's signing keypair and encapsulated in a ToBeEncrypted of type certificate_request_error. This is then encrypted with the response_encryption_key from the corresponding status request.

The resulting ciphertext is then encapsulated in a SymmetricEncryptedMessage as described in Section 5.3.5 RA → LCM (Status Confirm, success), and this in turn is encapsulated in a 1609Dot2Message of type symmetric_encrypted.

4 This field doesn't include start time, overlap, etc, as these are available within the individual SignedMessages.

The LCM shall verify the signature before taking action based on the error message. If the signature verifies, the LCM shall take the following action.

If the CertificateRequestError code is:

- csr_cert_verification_failure(0), the LCM shall regenerate and resend the status request
- csr_cert_revoked(2), the LCM shall log the error and stop requesting certs
- certificate_response_not_ready (7), the LCM shall regenerate and resend the status request after a configurable timeout period.
- (16), the LCM shall regenerate and resend the status request with the proper signature.
- invalid_request_hash(17), the LCM shall regenerate and resend the status request with the proper request hash.
- invalid_response_encryption_key (18), the LCM shall regenerate and resend the certificate request with a valid response encryption key.
- invalid_status(19), the LCM shall regenerate and resend the status request with the proper status.
- invalid_algorithm (20), the LCM shall regenerate and resend the status request with the proper algorithm.
- current_time_in_past(21) or current_time_in_future(22), the LCM shall regenerate and resend the status request with the proper current time.

NOTE: The LCM shall keep a list of all response_encryption_keys sent in the order in which they were sent. A response is considered "correct" if it decrypts correctly and the signature on the message inside the response verifies. If the LCM receives a correct response using a given response_encryption_key it shall discard any future messages encrypted with that response_encryption_key, to prevent replay attacks.

5.3.7 LCM → RA (Acknowledgement)

```
struct {
      opaque                    request_hash[10];
      opaque                    response_hash[10];
} ToBeSignedAnonymousCertResponseAck;
```

This type is used by the LCM to report to the RA the result of its processing of the certificate response. It shall be signed with the LCM's CSR cert, then encoded in a ToBeEncrypted of type anonymous_cert_response_ack, then encrypted with the RA's public encryption key, then encapsulated in a 1609Dot2Message of type encrypted.

- request_hash is the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedAnonymousCertRequestReq message contained in the request from the LCM to the RA

- The response_hash field is the low-order ten bytes of the hash of the encoded AnonymousCertRequestStatusCfm message contained in the response from the RA to the LCM.

If the RA receives an acknowledgement with an incorrect request_hash or response_hash value, or if the RA does not receive an acknowledgement, it shall log the error. This document does not define a mechanism for recovering from this error state.

5.4 Request Decryption Keys

5.4.1 Overview

- The LCM:
    - Generates a response_encryption_key, which is an AES-CCM key.
    - Generates a ToBeSignedAnonymousCertDecryptionKeyReq containing the information about the response encryption key and the cert batch for which it is requesting the decryption key.
    - Signs the ToBeSignedAnonymousCertDecryptionKeyReq with its CSR cert.
    - Encapsulates that in a ToBeEncrypted of type anonymous_cert_decryption_key_req.
    - Encrypts that with the RA's public encryption key.
    - Encapsulates that in an EncryptedMessage.
    - Encapsulates that in a 1609Dot2Message of type encrypted.
    - Sends that to the RA.

- The RA:
    - Extracts the EncryptedMessage
    - Decrypts the EncryptedMessage to recover the encapsulated SignedMessage
    - Verifies the SignedMessage (including checking that the CSR cert has not expired or been revoked)
    - Extracts the ToBeSignedAnonymousCertDecryptionKeyReq
    - Checks that the cert_batch_id corresponds to a cert batch that was issued to the requesting LCM (which is identified by its CSR cert).
    - If all of the above operations succeed, the RA:

        - Looks up the decryption key
        - Forms an AnonymousCertDecryptionKeyCfm
        - Encapsulates that in a ToBeEncrypted of type anonymous_cert_decryption_key_cfm.
        - Encrypts that with the response_encryption_key from the ToBeSignedAnonymousCertDecryptionKeyReq and forms a SymmetricEncryptedMessage containing the ciphertext, algorithm and key_id
        - Encapsulates that in a 1609Dot2Message of type symmetric_encrypted.
        - Sends to the LCM.
    - If any of the above operations fail:
        - Creates and signs a ToBeEncryptedCertificateRequestError as described in 1609.2 with the appropriate error code (see extensions to CertificateRequestErrorCode above)
        - Encrypts it with the response_encryption_key from the status request to form a SymmetricEncryptedMessage
        - Encapsulates the SymmetricEncryptedMessage in a 1609Dot2Message of type symmetric_encrypted.
        - Sends to the LCM.
- The LCM:
    - Decrypts the message
    - If successful:
        - Fills in a ToBeSignedAnonymousCertResponseAck.
        - Signs it with the CSR cert
        - Encapsulates it in a ToBeEncrypted of type anonymous_cert_decryption_key_ack.
        - Encrypts it with the RA's public key
        - Encapsulates the resulting EncryptedMessage in a 1609Dot2Message of type encrypted.
        - Sends to the RA.
    - If not successful, regenerates the response_encryption_key, and uses this as input to recreate and resend the request.

5.4.2 LCM → RA (Request)

```
struct {
      Time32                          current_time;
      CertificateProcessingStatus     status;
      SymmAlgorithm                   alg;
      uint8                           response_encryption_key[16];
      uint8                           cert_batch_id[8];
} ToBeSignedAnonymousCertDecryptionKeyReq;
```

This type is used to allow the sender to request the decryption key for an anonymous certificate batch. It shall be encapsulated within a 1609Dot2Message of type encrypted. The EncryptedMessage within the 1609Dot2Message shall be encrypted with the public key of the RA. When decrypted, the EncryptedMessage shall contain a ToBeEncrypted with type anonymous_cert_decryption_key_req and contents equal to a message of type AnonymousCertDecryptionKeyReq.

In this type:

- signer.type shall be certificate and signer.certificate shall be the CSR cert that the requester obtained on bootstrap.
- current_time is the time the status request was generated.
- status is the LCM's understanding of the status. It shall be:
    - first_request if this is the first status request with a given cert_batch_id value;
    - timeout if the LCM did not receive a response within the configurable timeout time to the last status request with this cert_batch_id value;
    - err_decrypt if the LCM got an error decrypting the response to the last status request with this cert_batch_id value or did not recognize the key_id in the SymmetricEncryptedMessage containing the response.
- alg shall identify the key used to encrypt the response. For this design it shall always be aes_128_ccm.
- response_encryption_key shall contain the AES key that shall be used to encrypt the response. This shall be different for every AnonymousCertDecryptionKeyReq from a given sender.
- cert_batch_id shall contain the relevant cert_batch_id value from the certificate response for which the LCM is requesting a certificate.
- signature shall be the signature, calculated over the encoding of handle, response_encryption_key and cert_batch_id, using the private key corresponding to the public verification key in signer.certificate.

5.4.3 RA → LCM (Confirm, success)

```
struct {
      uint8          cert_batch_id[8];
      SymmAlgorithm  alg;
```

```
      uint8          decryption_key[16];
} AnonymousCertDecryptionKeyCfm;
```

The type AnonymousCertDecryptionKeyCfm is used by the RA to return the decryption key for an anonymous certificate Batch.

The resulting ciphertext is then encapsulated in a SymmetricEncryptedMessage as described in Section 5.3.5 RA → LCM (Status Confirm, success), and this in turn is encapsulated in a 1609Dot2Message of type symmetric_encrypted.

When decrypted, the ciphertext of SymmetricEncryptedMessage shall contain a ToBeEncrypted with type anonymous_cert_decryption_key_cfm and contents consisting of a AnonymousCertDecryptionKeyCfm. This shall have fields set as follows:

- cert_batch_id is the relevant cert batch ID.
- alg is the symmetric algorithm, which shall always be aes_128_ccm.
- decryption_key is the decryption key for the algorithm identified in alg.

If a requester fails to receive an encrypted AnonymousCertDecryptionKeyCfm in a timely manner after sending an AnonymousCertDecryptionKeyReq, it shall regenerate and resend the AnonymousCertDecryptionKeyReq. The RA shall respond to a given AnonymousCertDecryptionKeyReq, sending the same AnonymousCertDecryptionKeyCfm every time.

5.4.4 RA → LCM (Confirm, failure)

```
    struct {
              SignerIdentifier             signer;
              opaque                       cert_batch_id[8];
              CertificateRequestErrorCode  reason;
              Signature                    signature;
          } ToBeEncryptedDecryptionKeyRequestError;
```

In the case of a failure the RA shall send the LCM an error message using the ToBeEncryptedDecryptionKeyRequestError structure. Here:

- signer.type shall be certificate and signer.certificate shall be the RA's certificate.
- cert_batch_id is the relevant cert batch ID.
- reason is the reason.

The ToBeEncryptedDecryptionKeyRequestError shall be signed with the RA's signing keypair and encapsulated in a ToBeEncrypted of type **anonymous_cert_decryption_key_error**. This is then encrypted with the response_encryption_key from the corresponding decryption key request.

The resulting ciphertext is then encapsulated in a SymmetricEncryptedMessage as described in Section 5.3.5 RA → LCM (Status Confirm, success), and this in turn is encapsulated in a 1609Dot2Message of type symmetric_encrypted.

The LCM shall verify the signature before taking action based on the error message. If the signature verifies, the LCM shall take the following action.

If the CertificateRequestError code is:

- csr_cert_verification_failure(0), the LCM shall regenerate and resend the decryption key request
- csr_cert_expired(1), csr_cert_revoked(2), csr_cert_unauthorized(3), request_denied(4), csr_cert_unknown (5), or canonical_identity_unknown (6), the LCM shall log the error and stop requesting certs.
- The values certificate_response_not_ready (7), cert_set_start_time_in_past (8), cert_set_start_time_too_far_in_future (9), cert_set_end_time_too_close_to_start_time (10), cert_set_end_time_too_far_from_start_time (11), requested_smaller_than_minimum_batch_duration (12), or requested_larger_than_maximum_batch_duration (13), are not valid in this case. If the LCM receives one of these values it shall log an error and resend the decryption key request. If, on continuing to resend, the LCM continues to receive one of these invalid values, it shall after a configurable number of times regenerate and resend the certificate request.
- The values requested_past_decryption_keys (14), requested_far_future_decryption_keys (15) the LCM shall log the message and go into an error state.

5.4.5 LCM → RA (Acknowledgement)

```
struct {
      uint8          cert_batch_id[8];
      uint8          decryption_key[16];
} ToBeSignedAnonymousCertDecryptionKeyAck;
```

The LCM shall send the RA an acknowledgement to confirm receipt using the ToBeSignedAnonymousCertDecryptionKeyAck type. It shall be signed with the LCM's CSR cert and placed in a SignedMessage, then encoded in a ToBeEncrypted of type anonymous_cert_decryption_key_ack, then encrypted with the RA's public encryption key, then encapsulated in a 1609Dot2Message of type encrypted. The fields have the identical values to the values in the corresponding AnonymousCertDecryptionKeyCfm.

If the RA receives an acknowledgement with an incorrect cert_batch_id or decryption_key value, or if the RA does not receive an acknowledgement, it shall log the error. This document does not define a mechanism for recovering from this error state.

5.5 Report Misbehavior

5.5.1 Overview

- The application submits a report to the LCM over an API to be determined.
- Upon receipt of a report from the application, the LCM :
    - Generates a response_encryption_key, which is an AES-CCM key.
    - Generates a ToBeSignedMisbehaviorReportReq containing the information about the response encryption key and the actual misbehavior report.
    - Signs it with its current anonymous certificate to generate a SignedMessage of (external) type misbehavior_report_req.
    - Encapsulates that in a ToBeEncrypted of (explicit) type misbehavior_report_req.
    - Encrypts that with the CA's public encryption key.
    - Encapsulates that in an EncryptedMessage.
    - Encapsulates that in a 1609Dot2Message of type encrypted.
    - LCM buffers 1609Dot2Messages and sends theseto the RA once a connection is available. 1609DotMessages are sent highest priority first, and within the highest priority newest report first.
- The RA:
    - Forwards message to CA.
- The CA:
    - Extracts the EncryptedMessage
    - Decrypts the EncryptedMessage to recover the encapsulated SignedMessage.
    - Sends a MisbehaviorReportAck, encrypted with the response_encryption_key from the ToBeSignedMisbehaviorReportReq.
    - Verifies the SignedMessage and checks that the anonymous certificate has not been revoked (it may have been expired though).
    - Extracts the ToBeSignedMisbehaviorReportReq.

- Analyzes the actual misbehavior report (or provides the report to a misbehavior report agent for analysis). Depending on the contents, the CA may choose to reject or accept a misbehavior report from a revoked unit.
- In case of detected misbehavior, collaborates with RA(s) and LAs to identify misbehaving entity. RA and CA update their CRLs.

- The RA:
  - Forwards message to LCM.
- The LCM:
  - Extracts the EncryptedMessage
  - Decrypts the EncryptedMessage with response_encryption_key to recover the encapsulated SignedMessage.
  - Verifies the SignedMessage with CA's certificate and extracts the ToBeSignedMisbaviorReportAck.
  - Validates the MisbehaviorReportAck and, if successful, deletes the 1609Dot2Message.

5.5.2 LCM → RA (Report)

```
struct {
      opaque                          misbehavior_report<var>;
      SymmAlgorithm                   alg;
      uint8                           response_encryption_key[16];
} ToBeSignedMisbehaviorReportReq;
```

This type is used to allow the sender to send a misbehavior report. It shall be signed with the anonymous certificate (short-term certificate if available, otherwise fall-back certificate) being valid at the time the LCM receives the misbehavior report, to create a SignedMessage of (external) type misbehavior_report_req. This shall be included within a ToBeEncrypted of (explicit) type misbehavior_report_req. This shall be encrypted with CA's public encryption key, obtained from the CA's certificate, to produce an EncryptedMessage. This in turn shall be encapsulated within a 1609Dot2Message of type encrypted.

In the SignedMessage:

- the signer.type field shall be certificate and signer.certificate shall be the anonymous certificate being valid at the time the LCM receives the misbehavior report.
- the signature field shall be the signature, calculated over the encoding of tbsmrr, using the private key corresponding to the public verification key in signer.certificate.

In the ToBeSignedMisbehaviorReportReq:

- The misbehavior_report field shall contain the misbehavior report. In this version of the specification, this will be the byte array representing one or more VSC3MisbehaviorReport messages (defined below). In other words, the misbehavior_report field contains one or more concatenated VSC3MisbehaviorReport messages.
- alg shall be aes_128_ccm.
- The response_encryption_key field shall contain the AES-CCM key to be used to encrypt the response. The LCM shall generate a fresh response encryption key every time and it shall not reuse response encryption keys.

```
enum {
      casual_report(0),
      alert_related_report(1),
      suspicious_message(2),
      … (255)
} MisbehaviorReportCategory;

struct {
      uint8                         version;
      ThreeDLocation                observation_location;
      Time64WithConfidence          observation_time;
      MisbehaviorReportCategory     misbehavior_report_category;
      1609Dot2Message               misbehavior_report_entries<var>;
} VSC3MisbehaviorReport;
```

In the VSC3MisbehaviorReport

- the version field shall contain a version determining the format of the misbehavior report. In this version of the specification, the version field shall be 1.
- the observation_location field shall describe the location of the reporter at the time of observation.
- the observation_time field shall describe the time of the reporter at the time of observation.
- the misbehavior_report_category field shall contain the category of the reported misbehavior, with possible categories defined in MisbehaviorReportCategory.
- the misbehavior_report_entries field shall contain one or more 1609DotMessages that are reported.

5.5.3 RA → LCM (Acknowledgement)

```
struct {
      opaque                    request_hash[10];
} ToBeSignedMisbehaviorReportAck;
```

This type is used by the CA to acknowledge receipt of the misbehavior report. It shall be signed with the CA's certificate, then encoded in a ToBeEncrypted of type misbehavior_report_ack, and then encrypted with the response_encryption_key from the

ToBeSignedMisbehaviorReportReq to form a SymmetricEnryptedMessage. This in turn is encapsulated in a 1609Dot2Message of type symmetric_encrypted.

- the request_hash field shall be the low-order ten bytes of the SHA-256 hash of the encoded ToBeSignedMisbehaviorReportReq message contained in the report from LCM to RA.

If the CA is not able to decrypt EncryptedMessage or verify SignedMessage, CA will not return an Acknowledgement nor error message.

If the LCM does not receive a ToBeSignedMisbehaviorReportAck within a configurable timeout, of if the LCM receives an incorrect request_hash, it shall resend the report.

If the LCM receives an incorrect request_hash, an invalid response, or no response three times, it shall stop sending the report.

5.6 Request CRL

5.6.1 Overview

- The LCM:
  - Generates a response_encryption_key, which is an AES-CCM key.
  - Generates a ToBeSignedCrlReq containing the type of request (Delta CRL or whole CRL) and, if required, the requested Delta CRL.
  - Signs it with its current anonymous certificate to generate a SignedMessage of (external) type crl_req.
  - Encapsulates that in a ToBeEncrypted of (explicit) type crl_req.
  - Encrypts that with the RA's public encryption key.
  - Encapsulates that in an EncryptedMessage.
  - Encapsulates that in a 1609Dot2Message of type encrypted.
  - Sends that to the RA.

- The RA:
    - Extracts the EncryptedMessage
    - Decrypts the EncryptedMessage to recover the encapsulated SignedMessage.
    - Verifies the SignedMessage and checks that the anonymous certificate has not been revoked.
    - Extracts the ToBeSignedCrlReq.
    - Validates the request.
        - If the request is invalid, creates a ToBeEncryptedCrlRequestError, signs that and encapsulates in a ToBeEncrypted. Then encrypts with the response_encryption_key from ToBeSignedCrlReq.
        - If the request is valid:
        - Locates the requested (delta or whole) Crl.
        - Encapsulates Crl in a ToBeEncrypted of type crl.
        - Encrypts with the response_encryption_key from the ToBeSignedCrlReq to form a SymmetricEncryptedMessage.
        - Encapsulates that in a 1609Dot2Message of type symmetric_encrypted.
- The LCM:
    - Extracts the SymmetricEncryptedMessage
    - Decrypts the SymmetricEncryptedMessage
    - In case of an error, handles the error.
    - Otherwise recovers the encapsulated Crl and processes the Crl.

5.6.3 LCM → RA (Request)

```
struct {
      CrlSeries                   crl_series;
      CertId8                     ca_id;
      uint32                      crl_serial;
      SymmAlgorithm               alg;
      uint8                       response_encryption_key[16];
} ToBeSignedCrlReq;
```

This type is used to allow the sender to request a CRL. It shall be signed with the current non-expired anonymous certificate to create a SignedMessage of (external) type crl_req. This shall be included within a ToBeEncrypted of (explicit) type crl_req. This shall be encrypted

with RA's public encryption key, obtained from RA's certificate, to produce an EncryptedMessage. This in turn shall be encapsulated within a 1609Dot2Message of type encrypted.
In the SignedMessage:

- the signer.type field shall be certificate and signer.certificate shall be the currently valid non-expired anonymous certificate.
- The signature field shall be the signature, calculated over the encoding of tbscr using the private key corresponding to the public verification key in signer.certificate.

In the ToBeSignedCrlReq:

- the crl_series field shall be the CRL series. In this specification, crl_series shall be 1.
- the ca_id field shall contain the CertId8 of the CA issuing the CRL, in this case the CAMP CA.
- the crl_serial field shall be set as follows:
    - if the whole CRL is requested, the field shall be set to 0xFF FF FF FF. Note: the whole CRL only includes entries that are not expired.
    - if the most recent delta CRL is requested without knowing the actual sequence number, the field shall be set to 0x7F FF FF FE.
    - If a specific delta CRL is requested, the field shall be set to the corresponding serial number. Allowed serial numbers are 0 to 0x7F FF FF 00.
- alg shall be aes_128_ccm.
- The response_encryption_key field shall contain the AES-CCM key to be used to encrypt the response. The LCM shall generate a fresh response encryption key every time and it shall not reuse response encryption keys

5.6.3 RA → LCM (Confirmation, success)

```
struct {
      CrlType         type;
      CrlSeries       crl_series;
      CertId8         ca_id;
      uint32          crl_serial;
      Time32          start_period;
      Time32          issue_date;
      Time32          next_crl;
      Select (type) {
      case (id_only):
          CertID10        entries<var>;
      case (id_and_expiry):
          IdAndDate       expiring_entries<var>;

// begin new material
      case (anonymous_entry):
          AnonymousEntry  anonymous_crl_entries<var>;
// end new material
```

```
      Unknown:
          Opaque           other_entries<var>;
      }
} ToBeSignedCrl;
```

A modified IEEE 1609.2/D9 structure ToBeSignedCRL of type anonymous_entry shall be used:

- type shall be anonymous_entry
- crl_series shall be the CRL series for which the CRL is used. In this specification, crl_series shall be 1.
- crl_serial shall be a counter that shall increment by 1 for every issued CRL by CA. For Delta CRLs, allowed crl_serial values are 0 to 0x7F FF FF 00. For full CRLs, allowed crl_serial values are 0x80 00 00 00 to 0xFF FF FF 00. There is a one-to-one mapping between Delta CRL with crl_serial dsr and Full CRL with crl_serial fsr such that
    - the Full CRL includes all entries of the Delta CRL and all previous Delta CRLs exclusive expired entries
    - fsr = dsr XOR 0x80 00 00 00.

  That is, the most significant bit of crl_serial is set to one for a Full CRL and set to zero for a Delta CRL, and the least significant 63 bits describe a sequential number.

```
struct {
      uint32   i;
      opaque   linkage_value_1[16];
      opaque   linkage_value_2[16];
      uint32   max_i;
} AnonymousEntry;
```

The type AnonymousEntry is used as CRL entry:

- the i field shall describe the time period i for which the linkage values are valid.
- the linkage_value_1 and linkage_value_2 field shall describe the revocation values at time period i.
- the max_i field shall describe the time period max_i until the CRL entry is valid.

CA issues a CRL by encapsulating ToBeSignedCrl in a Crl (including CA's signature).

RA shall encapsulate Crl in a ToBeEncrypted of type crl. This is encrypted using response_encryption_key from ToBeSignedCrlReq to form a SymmetricEncryptedMessage. This in turn shall be encapsulated in a 1609Dot2Message of type symmetric_encrypted.

5.6.4 RA → LCM (Confirmation, failure)

```
struct {
              SignerIdentifier            signer;
              opaque                      request_hash[10];
```

```
    CrlRequestErrorCode          reason;
    Signature                    signature;
} ToBeEncryptedCrlRequestError;
```

In the case of a failure the RA shall send the LCM an ToBeEncryptedCrlRequestError:

- signer.type shall be certificate and signer.certificate shall be the RA's certificate.
- request_hash shall be the the low-order ten bytes of the hash of 1609Dot2Message sent by the LCM.
- reason is the error reason.

The ToBeEncryptedCrlRequestError shall be signed with the RA's signing key and encapsulated in a ToBeEncrypted of type crl_req_error. This is then encrypted with the response_encryption_key from ToBeSignedCrlReq.

The resulting ciphertext is encapsulated in a SymmetricEncryptedMessage, and this in turn is encapsulated in a 1609Dot2Message of type symmetric_encrypted.

```
enum { verification_failure(0),
       anonymous_cert_invalid(1),
       anonymous_cert_revoked(2),
       request_denied(3),
       invalid_crl_series(4),
       invalid_crl_serial(5),
       invalid_ca_id(6),
       … (255)
     } CrlRequestErrorCode;
```

The LCM shall verify the signature before taking action. If the signature verifies, the LCM shall take the following action.

If the CrlRequestErrorCode is:

- verification_failure(0) - the LCM shall regenerate and resend the request.
- anonymous_cert_invalid(1) – the LCM shall use a valid anonymous certificate to sign the request.
- anonymous_cert_revoked(2) and request_denied(3) – the LCM shall log the error and stop requesting CRLs.
- invalid_crl_series(4) – the LCM shall regenerate and resend the request with a proper crl_series.
- invalid_crl_serial(5) – the LCM shall regenerate and resend the request with a proper crl_serial.
- invalid_ca_id(6) – the LCM shall regenerate and resend the request with a proper ca_id.

6. Model OBE Architecture

This section outlines a model architecture of the software running on the OBE, combining LCM and IEEE 1609.2 stack.

In a reference model, the LCM will have a defined interface which can be called by the 1609.2 stacks (LCM API). This API will provide certificates and singing keys to the 1609.2 implementations.

In turn, the LCM will expect functionality provided by the 1609.2 stacks (the 1609.2 stack API). These functions include parsing of 1609.2 messages, signing, verification, encryption and decryption calls. As there is no unified 1609.2 API, each 1609.2 stack needs to implement an adaptor that ensures the 1609.2 stack API can function correctly. A general overview is given in Figure 2.

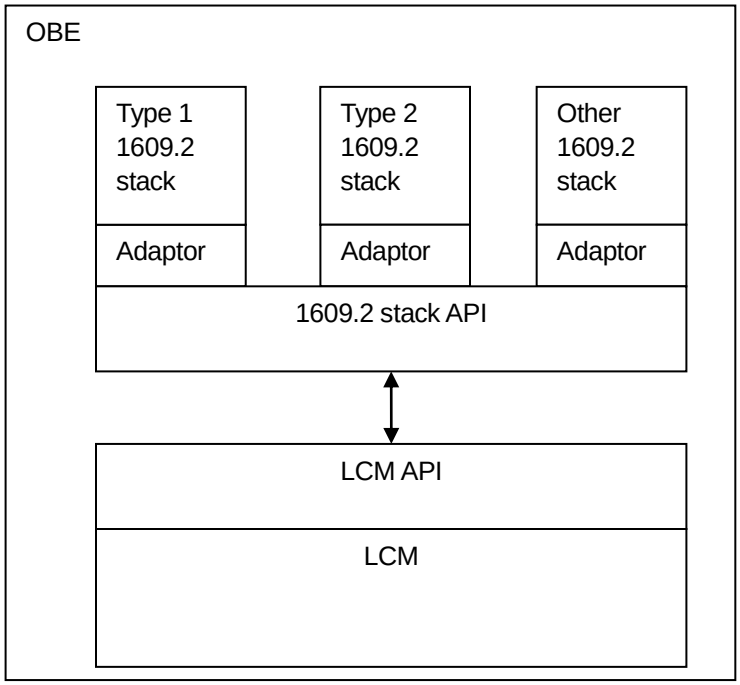


Figure 2: OBE Architecture

7. Model LCM Configuration Parameters

The parameters are described in Table 1: LCM configuration parameters. These parameters describe an exemplary set of configuration parameters.

Configuration Parameter	Default Value	Description
USE_ADDRESS		Indicates which Address type to use for CA: 1: Domain Name 3: IPv4 2: IPv6
CA_IP_V4_ADDRESS		
CA_IP_V6_ADDRESS	XXX.XXX.XXX.XXX	IP address of CA
CA_DOMAIN_NAME		
CA_TCP_PORT_NUMBER	16092	Port number of CA
PSID	???	PSID for which certificates are requested
SHORT_TERM_REQNEW	30000 [seconds] (500 minutes)	New certificates can be requested SHORT_TERM_REQNEW seconds before expiration of the current batch
SHORT_TERM_REQ_DCRYPTKEY	15000[seconds] (250 minutes)	New decryption key(s) can be requested SHORT_TERM_REQ_DCRYPTKEY seconds before available certificates expire.
BACKOFTIME	300[seconds] (5 minutes)	Time between status request messages
NAME	XXXX	Canonical name of the LCM
STORAGE_SPACE	1024 [kb]	Space available for certificate storage
BATCH_DURATION	2592000 [seconds] (30 days)	Length of a batch of certificates

Table 1: LCM configuration parameters

Sample configuration file:

```
**************************************************************************
#*                                                                        *
#* LCM sample configuration file                                          *
#*                                                                        *
#**************************************************************************
**/
```

```
# Network configuration parameters
RA_ADDRESS = localhost              # Servers host name or ip address
#RA_ADDRESS = 192.168.10.112        # Servers host name or ip address
RA_PORT = 1337                      # Servers port number

# PSID
PSID = 18                           # Range: 16-20 decimal
Storage_Space = 20480               # Certificate storage (kb)

#Bootstrap
Bootstrap_Request_Timeout = 30      # Seconds to wait for bootstrap
confirmation before requesting again

# Certificate
Batch_Duration_Units = 2            # Values: 0 = seconds, 1 = minutes, 2
= hours, 3= 60-hours, 4 = years
Batch_Duration_Value = 24           # 24 hr
Certificate_Request_Status_Inquiry_Interval =  70 # Minimum delay between
status inquiries, default: 70

Certificate_Request_Confirmation_Timeout =  5     # Seconds to wait for
confirmation before requesting again, default 5 sec
Decryption_Key_Request_Interval =  5              # Minimum delay between
decryption-key requests, default 5 sec

Maximum_Certificate_Storage_Time =  31536000      # 1 yr x 365 days/yr x
24 hr/day x 60 min/hr x 60 sec/min
Request_Certificates_Time =   30000               # 500 min. x 60 sec/min
Request_Decryption_Key_Time = 15000               # 250 min x 60 sec/min

# General
LCM_NAME = obe                      # Name to include in the certificate
request message
Connection_Retry_Interval =  60     # Minimum delay between connection
requests, defaul 60 seconds

# Logging options                   # Values: 0/1  Default: 0
LCMLogEnable = 0                    # Enable writting to the log
LogFileDirectory = .                # Name of directory for log files
LogUseSimpleName = 0                # Simple name: lcm.log

# The following options enable additional debugging information
LogEnableAdditionalInfo = 0         # Include lcm specific log messages

Log_Bootstrap_Request = 0           # Log bootstrap progress
Log_Bootstrap_Confirm = 0
Log_Bootstrap_Ack = 0
```

```
Log_CertRequest_Req = 0             # Log certificate request progress
Log_CertRequest_Confirm = 0
Log_CertStatus_Req = 0
Log_CertStatus_Confirm = 0
Log_CertResponse_Ack = 0

Log_DecryptKey_Request = 0          # Log decryption key progress
Log_DecryptKey_Confirm = 0
Log_DecryptKey_Ack = 0

Log_SignEncrypt_Input = 0           # Log data to be signed
Log_SignEncrypt_Before_Encrypt = 0  # Log data to be encrypted
Log_SignEncrypt_After_Encrypt = 0   # Log encrypted data

Log_Imported_File = 0               # log imported cert files
```

8. Security Server Architecture

8.1 Design Premise

The CA's ability to violate OBE's privacy is restricted by splitting Security Server roles into Registration Authorities (RA), Certificate Authorities (CA), and Linkage Authorities (LA). The minimum instantiation requires one RA, one CA, and two LAs. The minimum instantiation is implemented in this design but can be extended in future designs. The objective of the design is that no single authority is able to link certificates of an OBE (that is, if there are two certificates the likelihood of guessing whether the two certificates belong to the same OBE must not be larger than ½). For instance, LA1 and LA2 must collaborate to link certificates.

8.2 Overview

Figure 3 displays an overview of the Security Server design. It is assumed that the authorities (CA, RA, LA) communicate over secure connections (e.g., SSL protected).

8.3.1 Parser

The parser listens to a single port (TBD) waiting for client connections. Once a client becomes connected, a new thread is created and data from incoming OTA messages can be parsed.

8.3.2 RA

RA acts as anonymizer proxy: collect and process OBE requests, shuffle the requests, and forward individual certificate requests to CA.
RA also adds another layer of encryption: encrypts batches of certificates and provides decryption key upon request by OBE.

8.3.3 CA

CA issues certificates without knowing which certificate is for which OBE. The CA has a signature/verification key pair, S_SK_{CA} and S_PK_{CA} respectively, and an encryption/decryption key pair, E_PK_{CA} and E_SK_{CA} respectively.

8.3.4 LA

The LAs issue the linkage values (that are used as revocation values) and certificate IDs. LA1 and LA2 generate a set of CertIDs that are later combined by the CA.

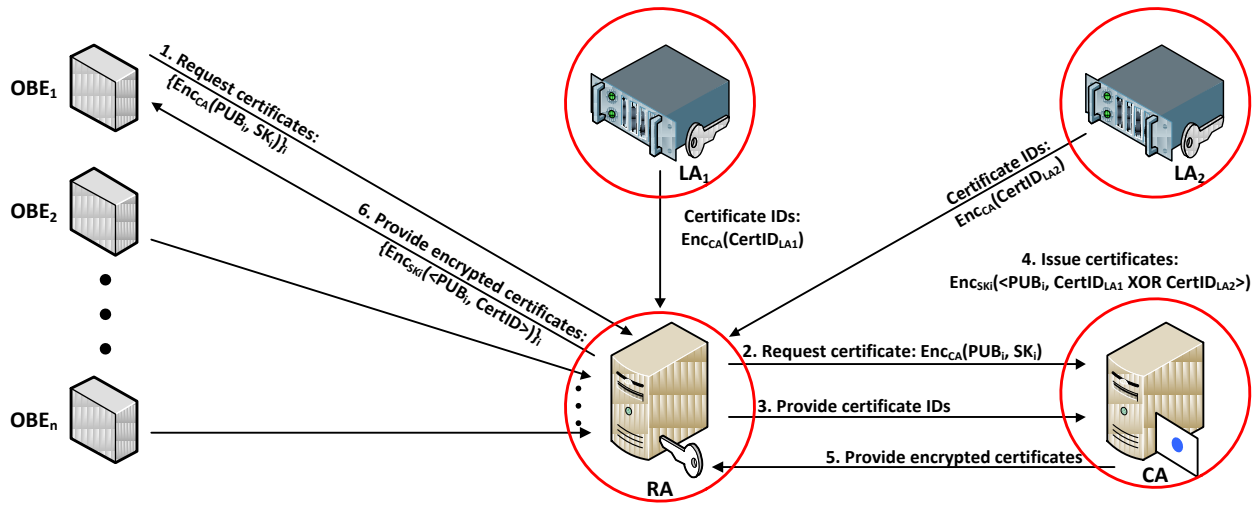
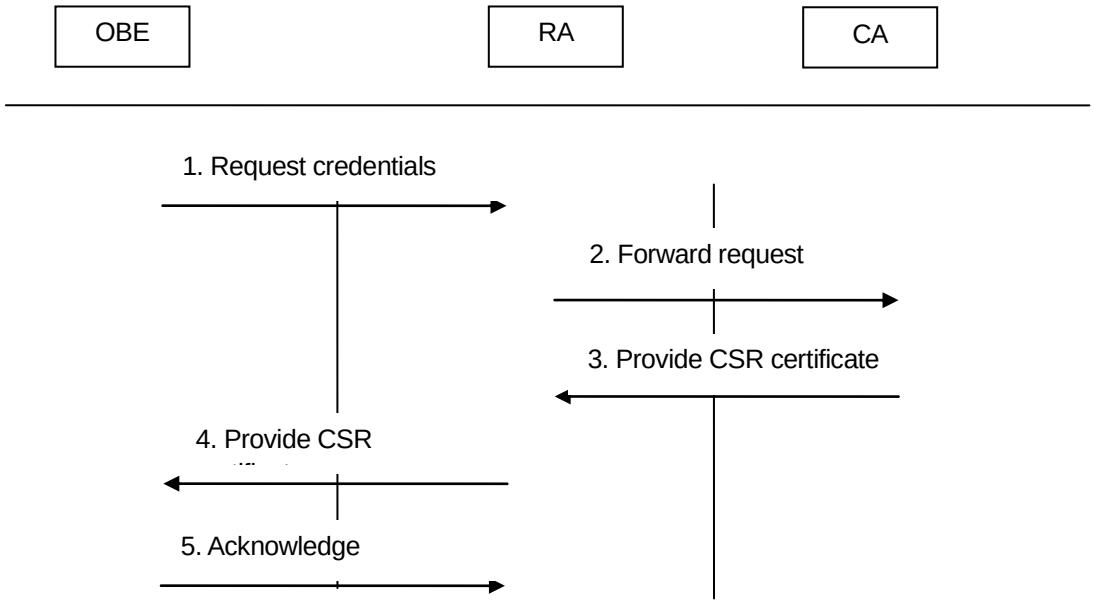


Figure 3: Security Server Design Overview

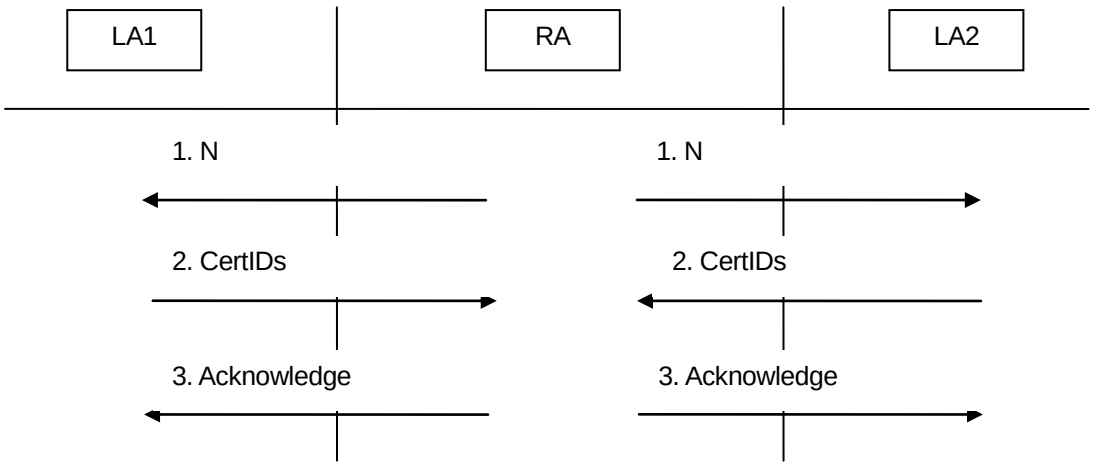
8.3 Bootstrapping

This process must be executed in a secure environment. For the given design, it is assumed that the process takes place in a secure environment.



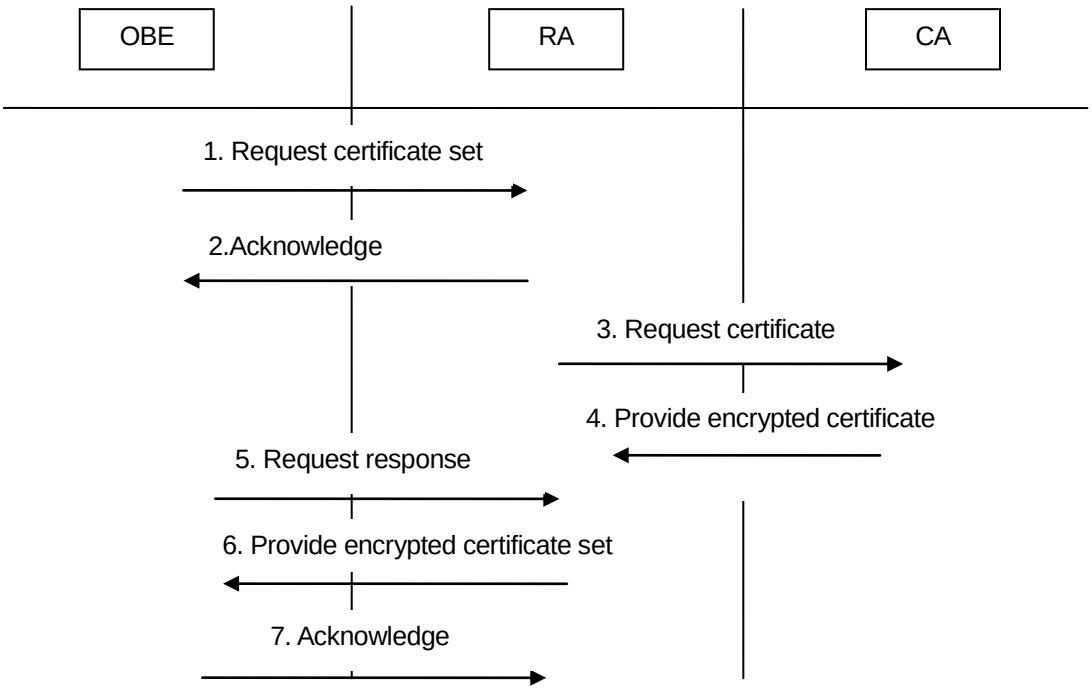
OBE	**RA**	**CA**
1. OBE → RA		
1.1 Generate seed key for implicit CSR certificate and pass to RA.		
2. RA → CA		
	2.1 Forward CSR request	
3. CA → RA		
		2.1 Issue CSR certificate and return together with $<PK_{RA}>_{CA}$ and $<PK_{CA}>_{CA}$ to OBE.
4. RA → OBE		
	4.1 Forward to OBE	
5.OBE → RA		
5.1 Acknowledge		

8.4 Calculation of Linkage Values



LA1	**LA2**
1. RA → LA1 / LA2	
1.1 RA provides a number of OBEs N for which linkage values should be generated	
2. LA1 / LA2 → RA	
2.1 Generate values s(w, i, 1) = hash(s(w, i-1, 1)) for OBEs *w*=1,...,N and *i*=1, …, n (n=365 to cover certificates for one year).	2.1 Generate values s(w, i, 2) = hash(s(w, i-1, 2)) for OBEs *w* and *i*=1, …, n (n=365 to cover certificates for one year).
2.2 Calculate CertID(w, i, j, 1) = $Enc_s(j)$ (with s = s(w, i, 1) and j=1, …, m) (m=288 to cover certificates for one day if each certificate is valid for 5:30 minutes with an overlap of 30 seconds). Note: w is not part of the calculation but is only used here to link together CertIDs.	2.2 Calculate CertID(w, i, j, 2) = $Enc_s(j)$ (with s = s(w, i, 2) and j=1, …, m) (m=288 to cover certificates for one day if each certificate is valid for 5:30 minutes with an overlap of 30 seconds). Note: w is not part of the calculation but is only used here to link together CertIDs.
2.3 Generate a set of fall-back certificates with j=1 and i being worth XXX years.	2.3 Generate a set of fall-back certificates with j=1 and i being worth XXX years.
2.4 Encrypt all values with E_PK_{CA} and sign the encrypted values.	2.4 Encrypt all values with E_PK_{CA} and sign the encrypted values.
2.5 Send the RA the tuples { Enc_{CA}(CertID(w, i, j, 1)), Sig_{LA1}(Enc_{CA}(CertID(w, i, j, 1))), w, i, j }for each w, i, j.	2.5 Send the RA the tuples { Enc_{CA}(CertID(w, i, j, 2)), Sig_{LA2}(Enc_{CA}(CertID(w, i, j, 2))), w, i, j }for each w, i, j.
2.5 Store tuples (Enc_{CA}(CertID(w, i, j, 1)), s(w, i, 1)).	2.5 Store tuples (Enc_{CA}(CertID(w, i, j, 2)), s(w, i, 2)).
3. RA → LA1 / LA2	
3.1 RA acknowledges receipt.	

8.5 Request Certificates



OBE	RA	CA
1. OBE → RA		
1.1 Generate ECC signing keypair a, A= aG, and h, H=hG. Randomly select values s and e (each 16 bytes). Collect current time *t,* start_time and end_time of short-term certificates, and (optionally) start_time_fallback and end_time_fallback of fall-back certificate, and certificate < R_PK_{OBE} >$_{RA}$. Sign [A, H, s, e, t, start_time, end_time,	1.2 Decrypt all values and verify the signature with R_PK_{OBE}. Check freshnessand validate R_PK_{OBE} against internal CRL.	

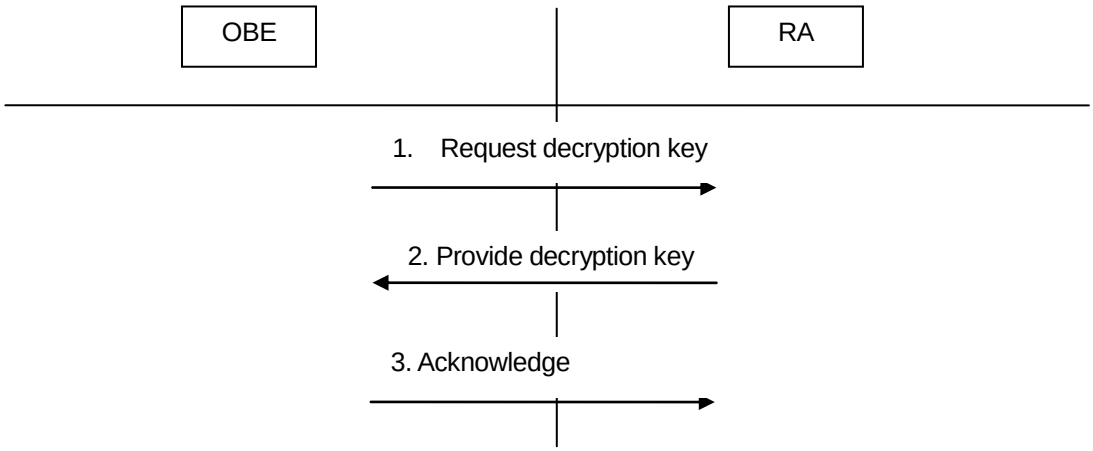
start_time_fallback, end_time_fallback, < R_PK_{OBE} >$_{RA}$] with R_SK_{OBE} and encrypt the signed message with PK_{RA}. Note: this step is highly flexible. Instead of start_time and end_time, the number of requested certificates might be included. Also PSID, geographic location scope, etc. might be included.		
	1.3 Decrypt message. Check signature. Store request as tuples (R_PK_{OBE}, start_time, end_time, certificate_type = short-term cert, decrypted_flag = false) and (R_PK_{OBE}, start_time_fallback, end_time_fallback, certificate_type = fall-back cert, decrypted_flag = true). Validate that OBE did not apply for certificates covering this time period before. Note: if more than one RA is used, the later list shall be shared by all RAs.	
2. RA → OBE		
	2.1 Acknowledge submission	
3. RA → CA		
	3.1 Assign an unused value w to the OBE. Request from LA1 the values Enc_{CA}(CertID(w, i, j, 1)) and request Enc_{CA}(CertID(w, i, j, 2)) from LA2 for each (i, j) (if RA requested a bundle from the LAs before, lookup values). Define the function f(k, i, j) as follows: a. Set I = J*i+j. Set J := 2^{32} b. Convert the integer I to an octet string by encoding I as an unsigned integer encoded in network byte order	3.2 The CA decrypts all values and verifies all signatures. The CA checks for each (encrypted) CertID (Enc_{CA}(CertID(w, i, j, 1)) and Enc_{CA}(CertID(w, i, j, 2)) that they were not used before. If they were used before, the CA rejects the request. The CA sets the contents of the cert to <CertID(w, i, j), $start\_time_n$,

	c. Set f equal to the octet string AES_k(0^128 XOR I) || AES_k(1^128 XOR I) d. Convert f to an integer assuming it was encoded in network byte order e. Reduce f modulo the order of the base point G. f. Output f. Note: due to the modulo operation, the output of f is slightly biased but impact is negligible. Then for each (i, j) (e.g. i=0, …, 364, j=0, …, 287 for one year) calculate and temporarily store • n = i*max(j) + j • $B_n = A + f(s, i, j) * G$ • $L_n = H + f(e, i, j) * G$ • $start\_time_n$ = start_time + n*cert_validity (for both short-term and fall-back certificates) • end_time_n = start_time_n + cert_validity (for both short-term and fall-back certificates) • RA stores the tuple (R_PK_{OBE}, w, i, j, $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))). Generate such values from a sufficient number of OBEs. Randomly select w, i, and j, and pass [Bn, Ln, $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))] to CA. Further information, including PSID, geographic location scope, etc. might be included. Note: the RA acts as an anonymity proxy. The RA must randomly pass individual requests to the CA not pass requests in bulk.	end_time_n>, the appropriate PSIDs, and further data (if any). The CA randomly chooses a value c mod the order of G. The CA then calculates • CertID(w, i, j) = CertID(w, i, j, 1) XOR CertID(w, i, j, 2) • $C = B_n + c*G$ • s = SHA-256(contents of cert, C) *c + S_SK_{CA} Finally, the CA ECIES encrypts the certificate and private key reconstruction value s using OBE's public key L_n as Enc_{Ln}(Cert contents, C, s) and signs this value as Sig_{CA}(Enc_{Ln} (Cert contents, C, s)). CA passes $SigEncCert_n$:= [Enc_{Ln}(Cert contents, C, s), Sig_{CA}(Enc_{Ln} (Cert contents, C, s)), $start\_time_n$, end_time_n] to RA. The CA stores the tuple (CertID(w, i, j), $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))).

4. CA → RA		
	4.1 RA collects all responses $SigEncCert_n$ for all n. The RA then generates batch identifiers $CertBatchID_u$, randomly generates an encryption key k_u for each $CertBatchID_u$, divides all encrypted certificates in batches where a batch has $start\_time_u$ (start_time of the first certificate) and end_time_u (end_time of the last certificate) and encrypts these as $EncBatch_u := \{Enc_{ku}(\{n, SigEncCert_n\}_n)\}_u$ (e.g., there could be 12 batches each worth 30 days of certificates, i.e. u=1…12, n=1…30*288). The RA stores the tuples $\{(CertBatchID_u, k_u, start\_time_u, end\_time_u)\}_u$ for all *u*.	
5. OBE → RA		
5.1 Request response		
6. RA → OBE		
	6.1 The RA passes the set of $\{CertBatchID_u, start\_time_u, end\_time_u, EncBatch_u\}_u$ for all u (e.g. u=1…12) to OBE. RA stores all values.	
7. OBE → RA		
7.1 OBE sends Acknowledge to RA.	7.2 RA deletes data that is not needed anymore.	

Note that the values of the requested certificates such as psid and geographic scope need to be checked for consistency and passed to the CA.

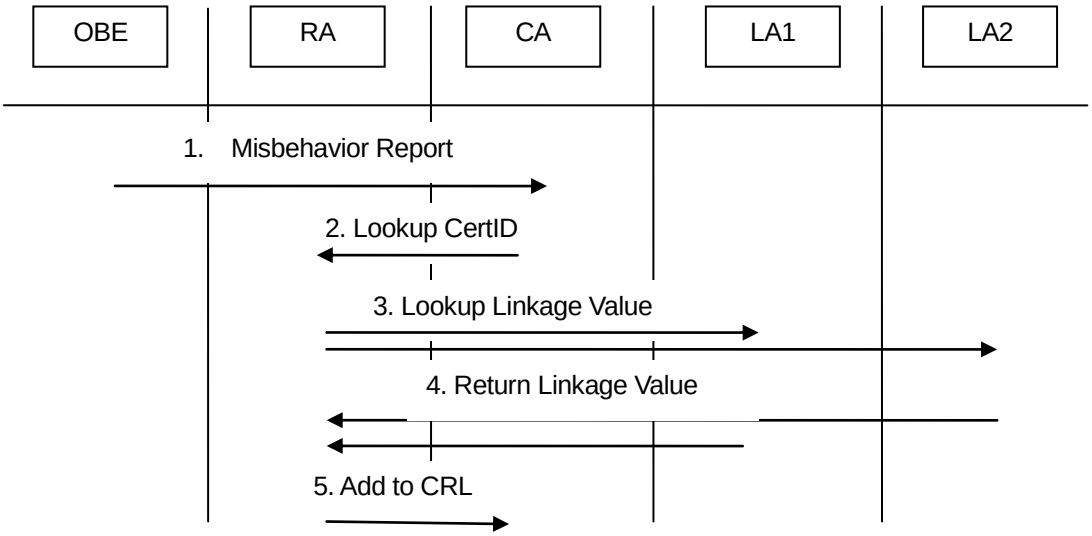
8.6 Request Decryption Key



OBE	**RA**
1. OBE → RA	
1.1 Select response encryption key *sk*, sign sk and $CertBlockID_u$ and certificate < R_PK_{OBE} >$_{RA}$ with R_SK_{RA}, and encrypt the signed message with R_SK_{OBE}.	1.2 Decrypt, verify signature, and validate against internal CRL.
2. RA → OBE	
	2.1 Lookup the tuple ($CertBlockID_u$, k_u, $start\_time_u$, end_time_u), check that current time fits to $start\_time_u$ and end_time_u, and encrypt $Enc_{sk}(k_u)$. Update table of provided certificates with tuple (R_PK_{OBE}, $start\_time_u$, end_time_u, certificate_type ={short-term cert, fall-back cert}, decrypted_flag = true). Provide $Enc_{sk}(k_u)$ to OBE.
2.2 Decrypt k_u and decrypt key block to recover values {n, $SigEncCert_n$ }$_n$ (for all values n of a block). Upon request, identify the proper value (n, $SigEncCert_n$) (e.g. based on $start\_time_n$ and end_time_n). Work out the appropriate range of i and j. Calculate	

• ECDSA verify $Sig_{CA}(Enc_{Ln}(Cert, c))$ (using CA's public key) • $l_n = h + f(e, i, j)$ (private encryption key) • ECIES decrypt Enc_{Ln}(Cert Contents, C, s) using l_n to recover the plaintext. • c_n = SHA-256(Cert Contents, C) * (a + f(s, i, j)) + s • Check that c_n*G = hash (Cert Contents, C) * C + S_PK_{CA}. c_n is the private key of *Cert*.	
3. OBE → RA	
3.1 OBE sends Acknowledgement	

8.7 Misbehavior Report and Revocation



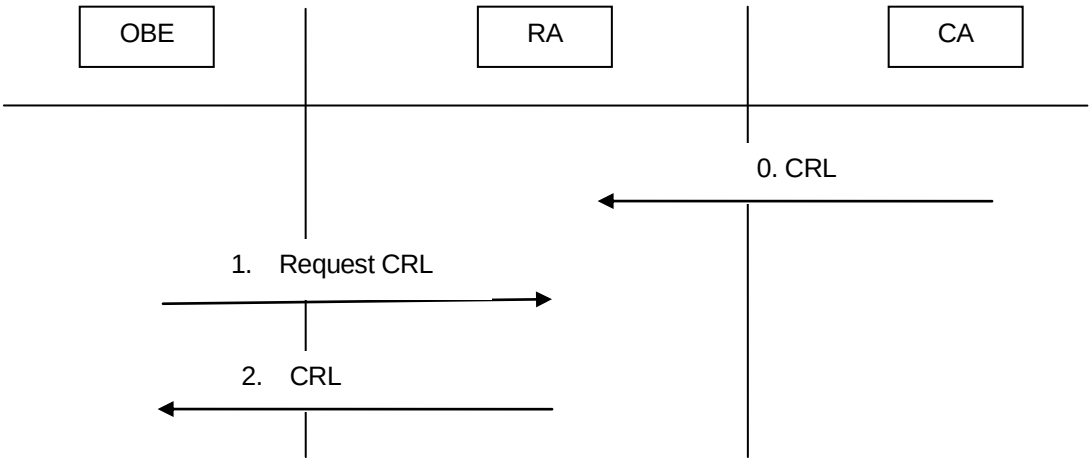
OBE	RA	CA	LA1	LA2
1. OBE → CA				
1.1 Assemble misbehavior report, sign with a message certificate Cert				

(with private key c_n), attach message certificate Cert, and encrypt with E_PK_{CA}.				
		1.3 Decrypt misbehavior report with SK_{CA} and verify signature with public key C_n recovered of Cert. Check if Cert is revoked (using public CRL).		
		1.4 Analyze misbehavior report. Identify certificate *mis_cert* of misbehaving vehicle (if any; mis_cert might be Cert of the reporter), and look-up tuple (CertID(w, i, j), $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))).		
2. CA → RA				
		2.1 CA passes Enc_{CA}(CertID(w, i, j, 1)) and Enc_{CA}(CertID(w, i, j, 2)) to RA.		
	2.2 RA searches for tuple (R_PK_{OBE}, w, i, j, $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))). If the found tuple points to a fall-back certificate, RA then searches the tuple for the current time that points to the short-term certificate (using search keys R_PK_{OBE} and w). Otherwise, RA searches the tuple for the current time that points to the fall-back certificate. Let the tuple be (R_PK_{OBE}, w, i', j', $start\_time_n$', end_time_n', Enc_{CA}(CertID'(w, i', j', 1)), Enc_{CA}(CertID'(w, i', j', 2))).			

	RA adds R_PK_{OBE} to internal CRL. RA then passes Enc_{CA}(CertID(w, i, j, 1)) and Enc_{CA}(CertID'(w, i', j', 1)) to LA1, and Enc_{CA}(CertID(w, i, j, 2)) and Enc_{CA}(CertID'(w, i', j', 2)) to LA2.			
3. RA → LA1, LA2				
			3.1 LA1 looks up s(w, i, j, 1) and s'(w, i', j', 1), and sends to RA.	3.2 LA2 looks up s(w, i, j, 2) and s'(w, i', j', 2), and sends to RA.
4. LA1, LA2 → RA				
	4.1 RA determines the validity period of certificates that are accessible by misbehaving OBE by looking up tuples with decrypted_flag = true, both for fall-back and short-term certificates: (R_PK_{OBE}, start_time, end_time, certificate_type ={short-term cert, fall-back cert}, decrypted_flag = true). Suppose the determined times furthest in the future are end_time_s (for short-term certificate) and end_time_f (for fall-back certificate). That is, the value end_time_s describes the time until the OBE was issued regular message certificates (until which decryption keys were provided), and end_time_f describes the time until fall-back certificates were issued. RA deletes all data about R_PK_{OBE} and forwards results			

	to CA.			
5. RA → CA				
		3.1 CA adds [(s(w, i, j, 1), s(w, i, j, 2), end_time$_s$, s'(w, i', j', 1), s'(w, i', j', 2), end_time$_l$] to public CRL. Note: it is assumed here that s refers to short-term certificate and s' refers to fall-back certificate. Implementer must order appropriately. The CA then signs the public CRL.		

8.8 Request CRL



OBE	RA	CA
0. CA → RA		
		0.1 provides CRL whenever available
1. OBE → RA		
1.1 Request CRL (signed and encrypted)		
1. CA → OBE		
	2.1 Respond with public CRL (encrypted)	

8.9 Keys

OBE	RA	CA	LA1 / LA2
• Global identity (long-term certificate) for requests: R_SK_{OBE} / $< R\_PK_{OBE} >_{CA}$	• RA encryption key pair: SK_{RA} / PK_{RA} • RA signing key pair: S_SK_{RA} / S_PK_{RA}	• CA key pairs: E_SK_{CA} / E_PK_{CA}, S_SK_{CA} / S_PK_{CA}	• LA signing key pair: S_SK_{LA} / S_PK_{LA}

8.10 Data Structures

OBE	RA	CA	LA1 / LA2
• secret values to expand private keys and decryption keys: a, h, s, e	• Time for which an OBE requested certificates: (R_PK_{OBE}, start_time, end_time, certificate_type ={short-term cert, fall-back cert}, decrypted_flag = {false, true}) • Internal CRL: List of [R_PK_{OBE}] • Encrypted (for CA) CertIDs: (R_PK_{OBE}, w, i, j, $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))) • Batch encryption keys: {$CertBatchID_u$, k_u, $start\_time_u$, end_time_u} for all *u*	• Public CRL: (List of revocation keys {s(w, i, j, 1), s(w, i, j, 2)}, Sig_{RA} • Mapping of CertIDs: (CertID(w, i, j), $start\_time_n$, end_time_n, Enc_{CA}(CertID(w, i, j, 1)), Enc_{CA}(CertID(w, i, j, 2))).	• Database to map encrypted CertIDs to revocation keys: (Enc_{CA}(CertID(w, i, j, 1/2)), s(w, i, 1/2))

9. Security Server Communication Socket

The Security Server will listen to a server socket on port GCM_TCP_portNumber. The messages transmitted over the TCP socket are expected in OTA message format (see Section 5. OTA Message Formats

) without any additional headers.

The details of establishing and closing sockets during the LCM – Server communication are explained in Appendix C: Connection requirements between LCM and Server.

10. Security Server Configuration Parameters

Table 2 describes the Security Server configuration parameters.

Table 2: Security Server Configuration Parameters

Configuration Parameter	Default Value	Description
GCM_T_shortTerm	330 [sec]	Int: Validity of short-term certificates
GCM_T_fallBack	94608030 [sec] (=3 years)	Int: Validity of fall-back certificate
GCM_T_overlap	30 [sec]	Int: Overlap of certificates
GCM_T_reload_shortTerm	2592000 [sec] (=30 days)	Int: New short-term certificates can be requested *T_reLoad_shortTerm* seconds before start time of requested batch.
GCM_T_reload_fallBack	64800000 [sec] (= 750 days)	Int: New fall-back certificates can be requested *T_reLoad_fallBack* seconds before start time of requested batch.
GCM_TCP_portNumber	1337	Int: TCP port of the Security Server.
GCM_T_request_decryptionKey	1296000 [sec] =(15 days)	Int: The decryption key for the next batch can be requested GCM_T_request_decryptionKey before expiration of the current batch.
GCM_EnableBootstrap	NO	Possible Values: YES, NO If set to 'NO', the server rejects any

		bootstrap request. If set to 'YES', the server will execute the bootstrap request.
GCM_StartEndTime_Max	31536000 [sec] (= 365 days)	Int: Maximum value allowed for LCM certificate request for difference between end_time and start_time
GCM_StartEndTime_Min	86400 [sec] (= 1 day)	Int: Minimum value allowed for LCM certificate request for difference between end_time and start_time
GCM_BatchDuration_Max	2592000 [sec] (=30 days)	Int: Maximum value allowed by LCM certificate request (batch_duration)
GCM_BatchDuration_Min	86400 [sec] (=1 day)	Int: Minimum value allowed by LCM certificate request (batch_duration)
CRL_IssueInterval	86400 [sec] (=1 day)	Int: the CA will issue a new CRL every CRL_IssueInterval seconds

11. Future Work

This section lists design options that were discussed during the project but left for future implementations:

- OTA error codes: instead of returning a single error code, a bit-map might be returned. There might be more than one appropriate error code, and the bit-map is able to indicate all applicable error codes
- The CA should have a way to communicate parameters such as min / max batch duration to the LCM
- The system should handle errors that arise during operations rather than during communications sessions with the CA. For example, when the LCM is decrypting an individually encrypted cert, any of the following can happen:
    - The CA signature on the encrypted cert could fail to verify
    - The encryption could fail
    - The CA signature check (or private key reconstruction and comparison with the derived public key, in the case of implicit certs) could fail

  We have not yet performed a threat analysis on these cases (for example, if the LCM notifies the PKI in this case, and the PKI issues a new set of certs, then a corrupt LCM can use this to get multiple sets of certs). We need to do this to work out the full process flow and then define messages.
- In Certificate Request, LCM – RA Acknowledgement, the error handling if no ACK is sent is very complex. It is worth analyzing whether an ACK is necessary at all. In any case, error handling needs to be carefully analyzed and designed.
- Error handling needs to be improved in implementations. Timeouts shall be defined in terms of time or repetitions.

12. References

[1] Anonymous, Cooperative Vehicle-To-Vehicle Crash Avoidance Applications Using 5.9 GHz, Dedicated Short Range Communications (DSRC) Wireless Communications, IP.com, document number IPCOM000210877D, dated September 14, 2011, www.ip.com

[2] IEEE Computer Society, *"IEEE Standard for Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 6: Wireless Access in Vehicular Environments,"* IEEE Std 802.11p™-2010.

[3] SAE International TM, *"Surface Vehicle Standard – Dedicated Short Range Communications (DSRC) Message Set Dictionary,"* SAE J2735, November 2009.

[4] Intelligent Transportation Systems Committee of the IEEE Society, *"Draft Standard for Wireless Access in Vehicular Environments – Security Services for Applications and Management Messages*," IEEE P1609.2TM/D9.3, 2011.

[5] DSRC Working Group of the Intelligent Transportation Systems Committee of the IEEE Society, *"Draft Standard for Wireless Access in Vehicular Environments (WAVE) – Networking Services,"* IEEE P1609.3 TM / REVCOM revision - TBD, August 2010.

[6] DSRC Working Group of the Intelligent Transportation Systems Committee of the IEEE Society, *"Draft Standard for Wireless Access in Vehicular Environments (WAVE) – Multi-channel Operation,"* IEEE P1609.4 TM / REVCOM revision - TBD, August 2010.

Appendix A: Certificate File Format

A.1 Private-Key-Certificate-File-Format

The following file format is designed to be a common format appropriate for the delivery of encrypted private keys and their associated certificates to various devices , whether short-lived certificates, long-lived certificates, fallback certificates, or certificates that do not expire. Besides being a common format, the goal is to provide fast access to a certificate with a particular start time.

The file contents will be identified by the file name. A concrete example is given in the next section for Vehicle Awareness Device short-lived certificates.

Different types of certificates for different devices may have different lengths and different validity periods, and therefore there may be a different number of certificates in a file. To accommodate these differences, this file format includes global data that defines the number of private-key-certificate entries in the file, and the time interval between the start times of successive certificates, which applies to each entry. A certificate that does not expire would be the only entry in the file, and in that case the time interval would be meaningless. Entries in the files will be arranged chronologically by the start times of the certificates. If an entry is missing for a particular starting time, that entry will have 256 arbitrary bytes, except that the ciphertext length byte will have the value 0.

In addition, each entry contains the length of the ciphertext for that entry, and each entry is padded so that it is 256 bytes long. Note that the padding may be arbitrary values. The fixed-length, chronological entries and the time interval between entries give the information required for fast access to a particular certificate.

Specifically, the file format is:

```
typedef struct {
    uint8_t  fileformat_version;      // file format version number
    uint8_t  num_entries[4];          // number of private-key-certificate
                                         entries in the file, big endian
    uint8_t  time_interval[4];        // number of seconds between the
                                         start times of successive
                                         certificates, big endian
    Entry    entry[num_entries];      // padded, encrypted private-key-
                                         certificate entries
} FileFormat;
```

The field `fileformat_version` contains the current version of the file format. The version described in this document is version 2, represented by the integer 2. Note `fileformat_version=1` is deprecated.

Each padded, encrypted private-key-certificate entry has the following format:

```
typedef struct {
    uint8_t nonce[12];                              // 12-byte nonce used to
                                                       AES-CCM encrypt this
                                                       entry
    uint8_t ciphertext_length;                      // number of bytes in
                                                       the ciphertext
    uint8_t ciphertext[ciphertext_length];          // ciphertext
    uint8_t padding[256-12-1-ciphertext_length];    // arbitrary padding
                                                       bytes
} Entry
```

After AES-CCM decryption (as specified in 1609.2) of the ciphertext in an entry, the resulting plaintext will have the following format:

```
typedef struct {
    uint8_t private_key_sign[32];           // private key for signing
    uint8_t private_key_dec[32];            // private key for decryption
    uint8_t encoded_certificate[ciphertext_length-16-32-32];
                                            // encoded 1609.2 Certificate
} PrivateKeyAndCertificate;
```

Note that the plaintext length is 16 bytes less than the ciphertext length because the ciphertext includes a 16-byte MAC, internally generated by AES-CCM encryption and removed during AES-CCM decryption.

Also note that the `private_key_dec` array contains 32 zero bytes if the certificate does not contain a public key for encryption and no decryption operation can be performed.

A.2 Short-Lived Certificate Files

In order to use a short-lived certificate, a Vehicle Awareness Device needs access to (a) the short-lived certificate (which includes a public verification key for messages signed using that certificate's private signing key) and (b) the private signing key associated with the public verification key in the short-lived certificate. 288 short-lived certificates (with associated private keys) are required to cover each day of operation. The LCDS will produce

a single file that contains the 288 short-lived certificates and associated private signing keys for that day and copy as many of these files as necessary to each SD card to cover the period when that SD card would be in use until it is replaced with a different SD card that contains certificates for an additional period.

The file names used to hold the certificates will follow the pattern "ShortLivedYYYYMMDD.crt" where YYYY is the 4-digit year, MM is the 2-digit month (e.g., "01" for January, "12" for December), and DD is the 2-digit day of the month (e.g., "01" for the first day of the month) for the day on which the certificates contained in the file are valid. (This date and all other times referred to in this document are UTC times. So, the file "ShortLived20110701.crt" would contain certificates valid from 7/1/2011 12:00:00 AM UTC until 7/2/2011 12:00:30 AM UTC.)

Within each file, the 288 certificates will be arranged in chronological order by the start time of the period during which each certificate is valid. So, the first certificate in the file ShortLived20110701.crt will be valid from 7/1/2011 12:00:00 AM until 7/1/2011 12:05:30 AM, the second certificate in this file will be valid from 7/1/2011 12:05:00 AM until 7/1/2011 12:10:30 AM, etc.

Each file will use the private-key-certificate file format specified above, where `num_entries` = 288 and `time_interval` = 5 min. * 60 sec./min. = 300 (seconds). A single key will be used for all devices and all device suppliers.

Because the encrypted private-signing-key/certificate entries are arranged chronologically with 256 bytes reserved for each item, the nth entry in the file is always associated with the nth time of the day. For example, the 7th entry is for the 7th time period of the day, so would be valid from 12:30:00 AM until 12:35:30 AM. (If a certificate is not available for a specific time period, the 256 bytes of the file reserved for that entry will be arbitrary except for the ciphertext_length which will be zero, thus ensuring that the nth entry is always associated with the nth time period of each day.)

A.3 RSE Certificate Files

In order to use an RSE certificate, an RSE needs access to (a) the certificate and (b) the corresponding private key. Within each file a single or small number of certificates will be arranged in chronological order by the start time of the period during which each certificate is valid.

The file names used to hold the WSA certificates will follow the pattern "WSAYYYYMMDD.crt" where YYYY is the 4-digit year, MM is the 2-digit month and DD is the 2-digit day of the month for the day of the start time of the first certificate in that file. The file names for TIM certificates will follow the pattern "TIMYYYYMMDD.crt", for SPAT "SPATYYYYMMDD.crt" ,for MAP/GID "GIDYYYYMMDD.crt" and for encryption certificates "ENCYYYYMMDD.crt" (used to send encrypted messages to CA/RA), respectively.

If an RSE certificate has a validity period of more than one day, the certificate file name reflects the first day on which that certificate is valid. If there is no new certificate on a particular day, there is no file named after that day.

Each file will use the private-key-certificate file format specified in Section A.1, where `num_entries` = 3 and `time_interval` = 10 months * 30 days/month * 24 hours/day * 60 min./hour * 60 sec./min. = 25920000 (seconds). A single key will be used for all devices and all device suppliers. This key will be different from the key used for short-lived certificates.

B.1 Overall

These are the security profiles for use in Safety Pilot Model Deployment only. This document does not constitute a commitment or indication as to the security profiles for use in any other project or deployment.

- Needs
    - All messages need to be signed so recipients can authenticate the source of the message. Recipients have the responsibility to determine if they trust the source.
    - Signatures will be checked on messages were action results. Messages collected in log files should be included regardless of authenticity.
    - The same cryptographic signature (and if needed encryption) process needs to be applied to all messages – BSM's, TIM's, SPaT's, GID's, WSA's, IP datagrams.

B.2 Security profile for BSM [5]

The following security profile shall be used for the SAE J2735 Basic Safety Message (BSM).

B.2.1 General

For each PSID: As defined in device specifications
Use1609Dot2 – true.

B.2.2 Secure messaging (sending)

Field	Value	Notes
SignMessages	True	
SetGenerationTimeInSecurityHeaders	True	The BSM does not contain sufficient time information to prevent replay attacks.
SetExpiryTimeInSecurityHeaders	False	BSM does not use expiry time
SetGenerationLocationInSecurityHeaders	False	The BSM message itself contains the generation location
SignerIdentifierType	Adaptive	If it has been half a second or more since a certificate was sent, attach a certificate to the message. Otherwise, attach a digest. A common root certificate will always be used. CAMP requires the OBE to support *SignerIdentifierType* =

[5] IEEE P1609.2™/D9 1 Draft Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages, May 2011.

		Certificate_Digest (1) and *SignerIdentifierType = Certificate (0)*
SignerIdentifierCertChainLength	-1 if used	
SignWithFastVerification	Compressed	
EncryptMessages	No	

B.2.3 Secure messaging (receiving)

Field	Value	Notes
VerifyMessages	Adaptive	A receiving OBE shall only verify a portion of received messages if the messages would result in an alert being raised to the driver based on an efficient strategy. A receiving RSE shall only verify messages if it would take some action other than logging as a result of those messages.
Check Validity Based on Generation Time	True	
GenerationTimeSource	Security Headers	
Check Validity Based on Expiry Time	False	Generation time is enough for BSM entities to judge relevance
ExpiryTimeOutsideSecurityHeaders	n/a	Expiry time is not used
Check Validity Based on Generation Location	False	In general, generation location is in message and location relevance check is carried out by receiving entity.
Generation Location Source	Message	Obtained from message.
AcceptEncryptedMessages	False	
DetectReplay	True	
MessageValidityPeriod	Adaptive	Configurable with default of 5s with a range of 1 to 120s.
MessageValidityDistance	n/a	BSM security does not use generation location – generation location is carried in the payload and processed by the entity
GenerationTimeConfidenceMultiplier	Adaptive	Default is 0, i.e. generation time confidence is ignored
OverdueCrlTolerance	Adaptive	Default is: CRL freshness checking not required.
SignWithFastVerification	Adaptive	Include compressed fast verification information if using explicit certificates.
ECPointFormat	Compressed	

Security management

Field	Value	Notes
SigningKeyAlgorithm	ECDSA-256[6]	
EncryptionAlgorithm	n/a	BSM does not use encryption

[6] Correction from listing in D9.

PublicKeyTransferType	Implicit	During initial testing, the Safety Pilot/Model Deployment Security Management System will generate a "test root CA certificate" and provide it to the vendors including the secret key. The vendors can use it to generate unit certificates, and to start interoperability testing at an early stage (before the official testing). Then, when available, the Safety Pilot/Model Deployment Security Management System will switch to the "real root CA certificate" slightly before the official testing (and do not provide the secret key to vendors).
ECPointFormat	Compressed	

B.3 Security profile for other Signed but NOT Encrypted Messages

The following security profile shall be used for the Safety Pilot/Model Deployment messages other than BSM's and WSA's that are signed but not encrypted – Signal Phase and Timing (SPaT), Geometric Intersection Description (GID/MAP), and Traveler Information Message (TIM).

B.3.1 General

For each PSID:
Use1609Dot2 – true.

B.3.2 Secure messaging (sending)

Field	Value	Notes
SignMessages	True	
SetGenerationTimeInSecurityHeaders	True	
SetExpiryTimeInSecurityHeaders	False	
SetGenerationLocationInSecurityHeaders	Adaptive	True for SPAT, False for GID/MAP,TIM.
SignerIdentifierType	Adaptive	If it has been half a second or more since a certificate was sent, attach a certificate to the message. Otherwise, attach a digest. . A common root certificate will always be used.
SignerIdentifierCertChainLength	-1 if used	Not used
SignWithFastVerification	Compressed	
EncryptMessages	No	

B.3.3 Secure messaging (receiving)

Field	Value	Notes
VerifyMessages	Adaptive	OBE will verify on state changes and as capacity allows during steady states.
Check Validity Based on Generation Time	True	

GenerationTimeSource	Security Headers	
Check Validity Based on Expiry Time	False	
ExpiryTimeOutsideSecurityHeaders	n/a	
Check Validity Based on Generation Location	Adaptive	True for SPAT, false for GID/MAP, for TIM
Generation Location Source	External	Security headers for SPAT, none for GID/MAP, for TIM.
AcceptEncryptedMessages	False	
DetectReplay	False	Replay of e.g. GID/MAP message is not an attack
MessageValidityPeriod	5s	Configurable
MessageValidityDistance	500m	Configurable
GenerationTimeConfidenceMultiplier	0, i.e. generation time confidence is ignored	Configurable
OverdueCrlTolerance	TBD – for November deliveries, don't check CRL overdueness	
SignWithFastVerification	Compressed	
ECPointFormat	Compressed	

B.3.4 Security management

Field	Value	Notes
SigningKeyAlgorithm	ECDSA-256	
EncryptionAlgorithm	n/a	
PublicKeyTransferType	Implicit	During initial testing, the Safety Pilot/Model Deployment Security Management System will generate a "test root CA certificate" and provide it to the vendors including the secret key. The vendors can use it to generate unit certificates, and to start interoperability testing at an early stage (before the official testing). Then, when available, the Safety Pilot/Model Deployment Security Management System will switch to the "real root CA certificate" slightly before the official testing (and do not provide the secret key to vendors).
ECPointFormat	Compressed	

B.4 Security profile for WME (Wave Service Announcements)[7]

The WME security profile shall be the following:

- SignWithFastVerification -- Compressed
- MessageValidityPeriod – 60 s

[7] IEEE P1609.2™/D9 1 Draft Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages, May 2011.

- MessageValidityDistance – 200m – in other words, WSAs from further away than 200m are rejected by the recipient
- GenerationTimeConfidenceMultiplier – 0
- RequiredCrlFreshness – Not checked
- SigningKeyAlgorithm – ECDSA-256
- EncryptionAlgorithm – None.
- PublicKeyTransferType – Implicit certificates
- ECPointFormat – Compressed

A WME shall sign a fresh WSA at the start of every minute, and shall continue to retransmit that WSA until the end of that minute.

B.4.1 Application security profile equivalents

In terms of an application security profile, the WSA security profile looks like this:

B.4.2 Secure messaging (sending)

Field	Value	Notes
SignMessages	True	
SetGenerationTimeInSecurityHeaders	True	
SetExpiryTimeInSecurityHeaders	True	Expiry time shall be 60s after generation time
SetGenerationLocationInSecurityHeaders	True	
SignerIdentifierType	Certificate_Chain	Send certificate with every transmission
SignerIdentifierCertChainLength	1	
SignWithFastVerification	Compressed	
EncryptMessages	No	

B.4.3 Secure messaging (receiving)

Field	Value	Notes
VerifyMessages	Adaptive	Verify a WSA the first time it is received; do not verify duplicates
Check Validity Based on Generation Time	True	
GenerationTimeSource	Security Headers	
Check Validity Based on Expiry Time	True	
ExpiryTimeSource	Security Headers	
Check Validity Based on Generation Location	True	
Generation Location Source	Security Headers	
AcceptEncryptedMessages	False	
DetectReplay	False	WSAs are meant to be replayed.
MessageValidityPeriod	60s	Configurable

MessageValidityDistance	500m	Configurable
GenerationTimeConfidenceMultiplier	0, i.e. generation time confidence is ignored	Configurable
OverdueCrlTolerance	TBD – for November deliveries, don't check CRL overdueness	
SignWithFastVerification	Compressed	
ECPointFormat	Compressed	

B.4.4 Security management

Field	Value	Notes
SigningKeyAlgorithm	ECDSA-256	
EncryptionAlgorithm	n/a	
PublicKeyTransferType	Implicit	During initial testing, the Safety Pilot/Model Deployment Security Management System will generate a "test root CA certificate" and provide it to the vendors including the secret key. The vendors can use it to generate unit certificates, and to start interoperability testing at an early stage (before the official testing). Then, when available, the Safety Pilot/Model Deployment Security Management System will switch to the "real root CA certificate" slightly before the official testing (and do not provide the secret key to vendors).
ECPointFormat	Compressed	

B.5 Security profile for Credential Management Messages (signed and encrypted)

Safety Pilot/Model Deployment IP messages used to interact with the Security Credential Management System shall use the certificate management messages defined in 1609.2 for identified certs and WSA certs, and shall use the certificate management messages defined by CAMP for anonymous certs. The encryption mechanism for credential management messages is explicitly described in 1609.2, Section 9. Security Server Communication Socket.

All communications and CSR certs shall be implicit certs using ECDSA-256.

Communications certificates used by RSEs in Model Deployment shall use the expiry field and shall have a lifetime of one year, with a start time randomly set to some point within the ten months preceding the start of Model Deployment (so most RSEs will change their certificate at some point within model deployment). This will apply to both WSA and WSMP certificates. CSR certificates used by RSEs in Model Deployment shall have an expiry date of June 30, 2014. Certificate lifetimes shall be encoded as (duration and expiry), in other words the flags use_start_validity and lifetime_is_duration shall be set.

Communications certificates used by RSEs in Model Deployment shall be of type identified (rather than identified_not_localized). They shall have a geographic scope in the form of a circular region, with center set equal to the operating location of the RSE and radius set equal to 10m.

RSE certificates shall not include an encryption key.

Each RSE shall have a single WSA certificate at any time; this WSA certificate shall contain the following list of PSIDs and priorities:
[INSERT LIST HERE]

WSA certs shall not use Service Specific Permissions (SSPs).

RSE certificates shall not use the from_issuer value anywhere.

All certificates shall have crl_series value equal to 1.

The subject_name field for each RSE certificate shall contain an ASCII string giving the unique name of that RSE device and the use of that certificate: for example "vendor1-rse1-SPAT", "vendor1-rse1-WSA". This name shall appear in the communications certificate, the CSR certificate, and each certificate request.

Certs used by OBEs, ASDs, and HIAs shall conform to the specification used by CAMP in its model deployment.

B.6 Certificates

B.6.1 Communications certificates

Each communications certificate used by a unit of any time shall have only one PSID.
Each CSR certificate used by a unit of any time shall have only one PSID.
Certificates will not include Service Specific Permissions (SSPs).

B.6.2 Certificate chains

There shall be no intermediate certificates and no certificate chains in the safety pilot model deployment. All certificates shall be issued directly by the root certificate.

B.6.3 Root certificate

There shall be a single root certificate. It shall have the following properties:

- It shall be an explicit certificate (version_and_type = 2)
- subject_type shall be root_ca.
- It shall have expiration time set to June 30, 2014 and crl_series set to 1.
- It shall not have the ContentFlag values use_start_validity or content_is_duration set.
- The signing key shall be an ECDSA-256 key given in compressed form.
- It shall have the ContentFlag value encryption_key set and contain an encryption key for ECIES-256 given in compressed form.
- The contents of the RootCaScope shall be:
    - Name: No name provided
    - Permitted_subject_types encoding: 83 ff (message_anonymous, message_identified_not_localized, message_identified_localized, message_csr, wsa, wsa_csr, message_ca, wsa_ca, crl_signer, message_ra)
    - Message Permissions: None (ie can authorize any PSID)
    - Wsa Permissions: None (ie can authorize any PSID and priority)

Geographic Region: None

Appendix C: Connection requirements between LCM and Server

This appendix clarifies requirements regarding the LCM establishing and closing connections with the server, according to the various phases of the message protocols.

C.1 Bootstrap

- The LCM shall establish a new connection with the server prior to sending the bootstrap request.
- The LCM shall maintain the connection waiting for the bootstrap confirm.
- After the LCM receives a bootstrap confirm message, it shall maintain the connection to send a bootstrap acknowledgement.
- After sending a bootstrap request, if the LCM receives a message other than a bootstrap confirm message, the LCM shall close the connection with the server.
- After sending a bootstrap request, if the LCM does not receive a bootstrap confirm message within its configured timeout period, the LCM shall close the connection with the server.
- The LCM shall use the currently open (bootstrap request) connection to send the bootstrap acknowledgement.
- After sending a bootstrap acknowledgement, the LCM shall close the connection with the server.

C.2 Certificate Request

C.2.1 Certificate Request

- The LCM shall establish a new connection with the server prior to sending an anonymous certificate request.
- The LCM shall maintain the connection waiting for an anonymous certificate request confirm.
- After the LCM receives an anonymous certificate request confirm, it shall close the connection with the server.

- After sending an anonymous certificate request, if the LCM receives a message other than an anonymous certificate request confirm, the LCM shall close the connection with the server.
- After sending an anonymous certificate request, if the LCM does not receive an anonymous certificate request confirm within its configured timeout period, the LCM shall close the connection with the server.

C.2.2 Status Request

- The LCM shall establish a new connection with the server prior to sending a status request.
- The LCM shall maintain the connection waiting for the status confirm.
- After the LCM receives a status confirm (success), the LCM shall maintain the connection to send a certificate response acknowledgement.
- After the LCM receives a status confirm (failure), the LCM shall close the connection with the server.
- After sending a status request, if the LCM receives a message that it cannot decrypt, the LCM shall close the connection with the server.
- After sending a status request, if the LCM receives a message other than a status confirm (success or failure), the LCM shall close the connection with the server.
- After sending a status request, if the LCM does not receive a status confirm (success or failure) within its configured timeout period, the LCM shall close the connection with the server.

C.2.3 Acknowledgement

- The LCM shall use the currently open (status request) connection to send a certificate response acknowledgement.
- After sending a certificate response acknowledgement, the LCM shall close the connection with the server.

C.3 Decryption-Key Request

- The LCM shall establish a new connection with the server prior to sending a decryption-key request.
- The LCM shall maintain the connection waiting for a decryption-key confirm.
- After receiving a decryption-key confirm (success), the LCM shall maintain the connection to send a decryption-key acknowledgement.

- After receiving a decryption-key confirm (failure), the LCM shall close the connection with the server.
- After sending a decryption-key request, if the LCM receives a message it cannot decrypt, the LCM shall close the connection with the server.
- After sending a decryption-key request, if the LCM receives a message other than a decryption-key confirm (success or failure), the LCM shall close the connection with the server.
- If the LCM does not receive a decryption-key confirm (success or failure) within its configured timeout, the LCM shall close the connection with the server.
- After sending a decryption-key acknowledgement, the LCM shall close the connection with the server.

C.4 Report Misbehavior

- The LCM shall establish a new connection with the server prior to sending a misbehavior report.
- The LCM shall maintain the connection waiting for a misbehavior-report acknowledgement response.
- After receiving a misbehavior-report response, the LCM shall close the connection with the server.
- After sending a misbehavior report, if the LCM receives a message it cannot decrypt, the LCM shall close the connection with the server.
- After sending a misbehavior report, if the LCM receives a message other than a misbehavior-report acknowledgement, the LCM shall close the connection with the server.
- After sending a misbehavior report, if the LCM does not receive a misbehavior-report acknowledgement within its configured timeout, the LCM shall close the connection with the server.

C.5 CRL Request

- The LCM shall establish a new connection with the server prior to sending a CRL request.
- The LCM shall maintain the connection waiting for a CRL confirm.
- After receiving a CRL confirm (success), the LCM shall close the connection with the server.

- After receiving a CRL confirm (failure), the LCM shall close the connection with the server.
- After sending a CRL request, if the LCM receives a message it cannot decrypt, the LCM shall close the connection with the server.
- After sending a CRL request, if the LCM receives a message other than a CRL confirm (success or failure), the LCM shall close the connection with the server.
- After sending a CRL request, if the LCM does not receive a CRL confirm (success or failure) within its configured timeout, the LCM shall close the connection with the server.

Appendix D: Registration process for ASDs

ASDs connect to the SCMS through the internet. This connection will be used to perform the bootstrap process in order to provide each ASD with a valid CSR certificate. It has to be ensured that only legitimate ASDs invoke the bootstrap process. ASDs will be required to identify themselves through the name field in the MessageCaScope of the BeSignedCertificateRequest that is used to request the CSR certificate. As per Section 5.2 this name has to be unique. A list of valid names will be given to each ASD supplier. Each entry of the list can be used by a single ASD to generate a valid bootstrap request. Each list entry can only be used once. Bootstrap requests that do not contain a valid name from the list will be rejected.

D.1 File Format

The list will be distributed in ASCII file format. Each line of the file will specify the name field in the MessageCaScope of the BeSignedCertificateRequest for a valid ASD. Each entry will have at least 80 bits of entropy (that is equivalent to 13 random ASCII characters/symbols).

U.S. Department of Transportation
ITS Joint Program Office-HOIT
1200 New Jersey Avenue, SE
Washington, DC 20590

Toll-Free "Help Line" 866-367-7487
www.its.dot.gov

[FHWA Document Number]

U.S. Department of Transportation

Research and Innovative Technology Administration